2-1-2008

Aspects of hardware methodologies for the NTRU public-key cryptosystem

Kyle Wilhelm

Aspects of Hardware Methodologies for the NTRU Public-Key Cryptosystem

by

Kyle Wilhelm

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in Computer Engineering

supervised by

Professor Marcin Łukowiak
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
February 2008

Approved by:

Prof. Marcin Łukowiak
Department of Computer Engineering

Prof. Stanisław Radziszowski
Department of Computer Science

Prof. Muhammad Shaaban
Department of Computer Engineering

Thesis Release Permission Form

Rochester Institute of Technology

Kate Gleason College of Engineering

Title: Aspects of Hardware Methodologies for the NTRU Public-Key Cryptosystem

I, Kyle Wilhelm, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Kyle Wilhelm

Date

Acknowledgments

I would like to thank my committee members, Dr.'s Łukowiak, Radziszowski and Shaaban for their advice and direction in completing my thesis. I would also like to thank Dr. William Whyte of NTRU Cryptosystems, Inc. for his assistance, I would not have progressed past the beginning of my work without his guidance and support.

The Computer Engineering department members have played an integral role in the completion of my degrees. I would not have gotten past my course work without the assistance of our head of department, Dr. Savakis. Our department is also blessed with an extraordinary staff, nothing would be possible without the help of Ms. Anne DiFelice, Mrs. Pam Steinkirchner, Mrs. Kathy Stefanik, Mr. Richard Tolleson and Mr. Charles Gruener.

Abstract

Cryptographic algorithms which take into account requirements for varying levels of security and reduced power consumption in embedded devices are now receiving additional attention. The NTRUEncrypt algorithm has been shown to provide certain advantages when designing low power and resource constrained systems, while still providing comparable security levels to higher complexity algorithms.

The research presented in this thesis starts with an examination of the general NTRUEncrypt system, followed by a more practical examination with respect to the IEEE 1363.1 draft standard. In contrast to previous research, the focus is shifted away from specific optimizations but rather provides a study of many of the recommended practices and suggested optimizations with particular emphasis on polynomial arithmetic and parameter selection. Various methods are examined for storing, inverting and multiplying polynomials used in the system. Recommendations for algorithm and parameter selection are made regarding implementation in software and hardware with respect to the resources available.

Although the underlying mathematical principles have not been significantly questioned, stable recommended practices are still being developed for the NTRUEncrypt system. As a further complication, recommended optimizations have come from various researchers and have been split between hardware and software implementations. In this thesis, a generic VHDL model is presented, based on the IEEE 1363.1 draft standard, which is designed for adaptation to software or hardware implementation while providing flexibility for changes in recommended practices.

Contents

Chapter 1

Introduction

At the most basic level, cryptography is used to allow two or more parties to communicate over an insecure channel without an unwanted party being able to interpret what is being transmitted [1]. To accomplish this goal, an intelligible message, or plaintext, is converted using some sort of encryption rule into an unintelligible message, or ciphertext. In order to obtain the original message from the ciphertext, a decryption rule is used to reverse the encryption rule. A key is used in both encryption and decryption to provide a common element for reversal. All of these elements combine to define a cryptosystem which contains [1]:

1. A finite plaintext space $\mathcal{P}$
2. A finite ciphertext space $\mathcal{C}$
3. A finite keyspace $\mathcal{K}$
4. A set $\mathcal{E}$ of encryption rules and a set $\mathcal{D}$ of decryption rules which allow, given a $K \in \mathcal{K}$, to define an encryption rule $e_K \in \mathcal{E}$ and decryption rule $d_K \in \mathcal{D}$. The rules $e_K : \mathcal{P} \rightarrow \mathcal{C}$ and $d_K : \mathcal{C} \rightarrow \mathcal{P}$ operate on each plaintext element $x \in \mathcal{P}$ such that $d_K(e_K(x))) = x$.

A cryptosystem can additionally be defined by cryptographic primitives and/or cryptographic protocols. A cryptographic primitive is a basic set of mathematical operations used to define a method of accomplishing a particular task. In the view of a full cryptosystem, the implementation specifics of encryption, decryption or the method to generate a key or keys could be considered a cryptographic primitive. A cryptographic primitive is

not intended to be secure by itself, but rather used as a building block to construct a full cryptosystem. Cryptographic protocols are descriptions of how cryptographic algorithms should be used. A cryptosystem is not necessarily secure if it is used in an improper way, so cryptographic protocols are defined to provide details on aspects such as key establishment or authentication.

1.1 Cryptanalysis

Cryptanalysis is the study of methods used to obtain private information, most frequently to obtain the key used for decrypting messages, through examination of available information, usually ciphertexts. In early cryptographic systems, little or no attempt was made to disguise the operation of encryption. For a time when literacy was not widespread, this could be considered a valid approach assuming that anyone intercepting an encrypted message was most likely not competent enough to decrypt it. As education and publications became available to a larger audience, efforts were made to establish certain aspects for cryptography that would allow for the analysis of how secure a method or system would be. Auguste Kerckhoffs developed a list of principles, including the idea that a cryptosystem should be secure even if the details of the system are publicly available, excluding the secret key [2]. To establish an idea of how secure a system is, the concept of defining the amount of computational power needed to break a system was developed. Although Claude Shannon was able to prove that the one-time pad cipher is unbreakable under the proper conditions [3], the one-time pad cipher is considered impractical to implement due to the difficulty of generating random numbers and the risks involved in distributing the long keys needed for the cipher. Modern systems implement much shorter keys and use computational effort based on the best known attack against the system as a measure of security. To evaluate a system, a certain set of assumptions are applied including an attack model. Attack models define the type and amount of information available to an attacker. Common attack models include the ciphertext only attack, the known plaintext attack, the chosen ciphertext attack

and the chosen plaintext attack. In the ciphertext only attack, the attacker has access to a ciphertext. A known plaintext attack is one in which a plaintext and the corresponding ciphertext are known to the attacker. The chosen plaintext attack assumes the attacker has access to the encryption method to input chosen plaintexts and receive the corresponding ciphertexts. Gaining access to the decryption method in order to input chosen ciphertexts and obtain the corresponding plaintexts is the chosen plaintext attack. In most cases, the worse case attack model is used for a particular system in order to generate a minimum known security level.

1.2 Private-key cryptography

Symmetric-key or private-key cryptography systems use identical or simplistically related keys for encryption and decryption. In a case where a secure channel exists, a private key can be communicated between parties. Subsequent communication can proceed on an insecure channel without comprising data under the assumption that no other party has the private key available to decrypt messages. Private-key cryptographic systems balance the requirement of securely communicating the private key with encryption and decryption processes that require comparatively low computational effort. Cryptanalysis of a private-key system will attempt to recover the private key through analysis of message data or perhaps physical system behavior.

1.3 Public-key cryptography

Public-key cryptography, first introduced in 1976 by Diffie and Hellman [4], is a scheme used to securely communicate over insecure channels without a previously established secure channel. Secure transmission is established through the use of a public and private key. Each party retains a non-shared private key and shares their public key with users who wish to communicate with them. The sender then encrypts messages using the receiver's

public key and the receiver decrypts the messages using their private key. Public-key cryptographic systems balance the ability to communicate on insecure channels with higher complexity encryption and decryption methods that are assumed to be computationally infeasible to break. Due to the public key being assumed to be widely available, a public-key cryptosystem should be constructed such that information about the private key cannot be determined using information about the public key. An array of distribution issues come with publicly available information. Authentication of ownership is needed to establish that a particular public key belongs to a trusted source. Digital signature algorithms have been developed to aid in the authentication of information. Key agreement protocols are used to establish a key between two or more parties, allowing each party to participate in generation of a common key. Key distribution protocols define distribution methods for a trusted source to transmit key information to the relevant sources. Cryptanalysis of a public-key system may also attempt to recover the private key, but additional information may obtained through analysis of the public key which would aid in recovery of the private key. Due to security of public-key systems being dependent on an underlying mathematical problem, security can also be comprimised by researching faster, more efficient methods to solve the underlying problem.

1.3.1 Classical public-key cryptosystems

A certain subset of public-key cryptosystems can be termed classical in the sense that they are based on widely known and deeply studied hard problems. Public-key systems based on the integer factorization problem, discrete logarithm problem or elliptic curve discrete logarithm problem fall into this category. Integer factorization based systems rely on the difficulty of factoring a large composite number into its prime factors. The RSA cryptosystem, named after the initials of its authors, uses the integer factorization problem combined with modular exponentiation to achieve encryption and decryption [5]. Cryptosystems based on the discrete logarithm achieve security through the difficulty of finding an exponent in a cyclic group to satisfy a chosen equation. More specifically, the discrete logarithm

problem can be defined by the following.

- Given a multiplicative group $(G, \cdot)$ and an element $\alpha \in G$ of order n, define:

$$\alpha\rangle = \{\alpha^i \colon 0 \leq i \leq n-1\}, \text{ and let } \beta \in \langle\alpha\rangle$$

- Find the unique integer a such that:

$$\alpha^a = \beta \text{ and } 0 \leq a \leq n-1$$

- Alternatively, $a = \log_\alpha \beta$

The ElGamal cryptosystem is based on the discrete logarithm problem, achieving security through the difficulty of finding the unique exponent, the properties of the underlying cyclic group and protection against semantic attacks [6]. The elliptic curve discrete logarithm problem is similar to the discrete logarithm problem but uses scaling of a point on an elliptic curve. The system space is defined by a finite field, an odd prime field or power of two field, in which an elliptic curve is defined by the points in the field which satisfy a chosen cubic equation. By choosing a point G, a cyclic group can be defined by the scalar integer multiples of that point, $(0, G, G+G, G+G+G, ...)$. By choosing a point G and calculating the scalar multiplication result kG, the hard problem is to calculate k given G and kG. Many variations of elliptic curve cryptosystems exist, the original suggestion being posed independently by both Koblitz [7] and Miller [8].

1.4 Application of public-key cryptography

Although public-key cryptography is relevant in both wired and wireless applications, the requirements are quite different. A general assumption that can be made for wired application of cryptography is that the amount of security desired is the driving factor for the cryptosystem. In this case, the cryptosystem is allowed to use any required resources for any required amount of time, within practicality, to accomplish the desired security level. The increasing number of applications which transmit confidential data over insecure channels requires that public methods be made available for authentication and transmission of data between sources. After contact is established, it is frequently preferred to use public-key

cryptographic methods to communicate a private key for use by both parties in a symmetric cryptosystem for the sake of efficiency. From a wired perspective, a public-key method that approaches the efficiency of symmetric schemes may allow for communication using public-key methods only.

In wireless applications, the desired level of security must be balanced with the amount and availability of computational resources. Due to the lack of computational and memory resources, early assumptions were that private-key cryptography would be required. Since a secure channel cannot be assumed to be available, alternative schemes have been developed to implement private-key cryptosystems on an insecure channel, but they require complex protocols to accomplish [9] and have been shown to be vulnerable to certain types of attacks [10]. Public-key cryptosystems would seem to provide desired security services as well as provide security advantages in the event that a node in a wireless network is compromised [11]. Current trends seek to allow wireless computing in an embedded environment, requiring restriction of the amount of computational power, memory storage, gate area and power that are allowed to be consumed by such devices. Existing public-key schemes have been found to be challenging in terms of resource consumption (e.g., El Gamal, RSA) or in terms of power scalability (e.g., ECC) [11].

Chapter 2

NTRU background

2.1 NTRU Public-key Cryptosystem

The NTRU Public-Key Cryptosystem (PKCS) was developed due to the interest in computationally fast and efficient methods to implement public-key cryptography. The encryption process is accomplished through polynomial ring arithmetic modulo two relatively prime values (integer or polynomial). The decryption process reverses the encryption process using probabilistic methods with a chance of failure, although this is minimized or eliminated through selection of the system parameters. The choice of system parameters, as with most cryptosystems, is the main method for determining the projected level of security [12].

2.1.1 Convolution polynomial rings

The primary NTRU operations are performed on polynomials of degree $N-1$ with integer coefficients in a convolution polynomial ring. The use of the term convolution polynomial ring refers to symbolic reduction of the polynomials used in the system. Addition in the polynomial ring is performed as with standard polynomials, coefficients of equal degree are added together. Multiplication is also performed the same as with standard polynomial except for the additional rule that $X^N \equiv 1$. The result of this rule is that X^N is replaced by 1, X^{N+1} is replaced by X and so on. The general computation method is that the k^{th} coefficient c_k is the dot product of the coefficients of a with the coefficients of b, where the

coefficients of b are reversed and rotated $k+1$ positions [12].

$$a + b = (a_0 + b_0) + (a_1 + b_1)X + ... + (a_{N-1} + b_{N-1})X^{N-1}$$
$$a * b = c_0 + c_1X + c_2X^2 + ... + c_{N-2}X^{N-2} + c_{N-1}X^{N-1}$$
$$c_k = a_0b_k + a_1b_{k-1} + ... + a_kb_0 + a_{k+1}b_{N-1} + a_{k+2}b_{N-2} + ... + a_{N-1}b_{k+1}$$

Often, the notation used to represent that operations in the NTRU occur in a convolution polynomial ring is denoted as $\mathbb{Z}[X]/(X^N - 1)$, where $\mathbb{Z}[X]$ represents a polynomial with integer coefficients and the division $(X^N - 1)$ represents symbolic reduction of the polynomial.

2.1.2 Basic parameters

The most basic parameters of the NTRU cryptosystem are N, p, and q. The parameter N is used to define the degree of polynomials used in the convolution polynomial ring. The modulus p is defined as the small modulus and the modulus q is the large modulus, where p is much less than q. Most operations in the convolution ring will occur modulo q whereas the modulus p is used to reduce the random generation components used in encryption and to constrain the message space [12]. Modular reduction of a convolution polynomial is performed by reducing each coefficient.

2.1.3 NTRU key generation

The key generation scheme is used to generate the private and public key pair. The process begins by choosing two small polynomials f and g, where small is defined as having coefficients much smaller than the large modulus q. The inverse of f is calculated both modulo p and modulo q, generating $f_p * f \equiv 1 \pmod{p}$ and $f_q * f \equiv 1 \pmod{q}$. The values f and f_p are retained as the private key pair and the public key h is calculated using p, f_q and g [12].

$$h \equiv pf_q * g \pmod{q}$$

2.1.4 NTRU encryption

The encryption process starts by generating a polynomial message m whose coefficients lie in an interval of length p, which is normally centered around zero. A small random blinding polynomial, r, is then generated and used to obscure the message. The final encryption uses m, r and the public key h to generate e, the encrypted message [12].

$$e \equiv r * h + m \pmod{q}$$

2.1.5 NTRU decryption

The decryption process first uses the private key f to calculate:

$$a \equiv f * e \pmod{q}$$

The coefficients of a must be chosen in the proper interval of length q to ensure the highest probability that the decryption process will be successful. Once the coefficients of a are chosen on the proper interval, a is reduced modulo p and the second private key is used to compute:

$$b \equiv a \pmod{p}$$
$$c \equiv f_p * b \pmod{p}$$

If decryption has successfully completed, then the polynomial c will be equal to m, the original message [12].

2.1.5.1 Why decryption works

To understand the decryption process, an understanding of the reason behind the decryption steps must first be reached. During the encryption process, polynomials modulo the smaller modulus p are combined to form polynomials modulo the larger modulus q. In this case, the results of reduction are not as critical because the smaller modulus space is contained within the larger modulus space. During decryption, the process of encryption is reversed,

meaning the polynomials are being constrained from the larger modulus space back to the smaller modulus space. The operations must be examined carefully because a reduction in the larger modulus space can cause a change in the residue class of the reduction using the smaller modulus, which would cause the recovered message to differ from the original message. In order to get a better idea of the problem that can occur, an example below is provided using the product of 34, 5 and 7 reduced both modulo 37 and modulo 3 in a number of cases.

- Case 1: Reduction of 1190 using the modulus 3

$$1190 \equiv 2 \pmod{3}$$

- Case 2: Reduction of 1190 using the modulus 37

$$1190 \equiv 6 \pmod{37}$$

- Case 3: Reduction of 1190 using the modulus 37 followed by reduction using the modulus 3

$$1190 \equiv 6 \pmod{37}$$
$$6 \equiv 0 \pmod{3}$$

From comparison of the first case and the third case, an example can be seen where the result of reduction in a larger modulus can change the result of reduction in a smaller modulus. A possible solution for this issue is to eliminate a factor in the product to reduce result of the product to be less than the larger modulus. In the example provided, if the factor 34 were eliminated, then the product would be 35 and reduction modulo the larger modulus 37 would have no effect. The decryption process for NTRU uses this concept, except that the operations occur on polynomials with integer coefficients.

Decryption makes use of the public key h, the private key pair f and f_p, and the encrypted message e. The encrypted message is the sum of the original message and scaled version of the public key, $e \equiv r * h + m \pmod{q}$. Out of the parameters, the receiver only has the value of h and therefore has to use mathematical methods to retrieve the message. To get a better idea of what the encrypted message is equivalent to, a substitution for the public key can be made, $h \equiv pf_q * g \pmod{q}$. The result allows for analysis of the

components of the encrypted message, $e \equiv r * (pf_q * g) + m \pmod{q}$. The polynomials r, g and m and modulus p are all comparatively small to the modulus q. To eliminate the large factor, f_q, the encrypted message can be multiplied by the private key f. Note that all calculations are performed within the convolutional polynomial ring, an implicit reduction modulo $X^N - 1$ occurs before reduction modulo either p or q.

$$e \equiv r * h + m \pmod{q}$$
substitute $h \equiv pf_q * g \pmod{q}$
$$e \equiv r * (pf_q * g) + m \pmod{q}$$
multiply by the private key f
$$a \equiv f * e \equiv f * (r * pf_q * g) + f * m \pmod{q}$$
since $f * f_q \equiv 1 \pmod{q}$*, the result is*
$$a \equiv f * e \equiv pr * g + f * m \pmod{q}$$

The private key f is small compared to the modulus q as well, so the result a is comprised of polynomials and a scalar which are all comparatively small to q. With the removal of f_q and the proper choice of system parameters, the value of a should be unchanged whether it is modulo q or not. The consequence of this is that not only is $a \equiv pr * g + f * m \pmod{q}$ but more importantly $a = pr * g + f * m$. To finish retrieving the original message, the result a is reduced modulo p and multiplied the second private key f_p.

$$a \equiv pr * g + f * m \pmod{q}$$
reduce modulo p
$$b \equiv pr * g + f * m \pmod{p}$$
the quantity $pr * g$ *is reduced to zero*
$$b \equiv f * m \pmod{p}$$
multiply the private key f_p
$$c \equiv f_p * b \equiv f_p * f * m \pmod{p}$$
since $f * f_p \equiv 1 \pmod{p}$*, the result is*
$$c \equiv m \pmod{p}$$

The original message was constructed modulo p, so c is exactly equal to the original message m assuming all of the operations were successful.

2.2 Standards

NTRU Cryptosystems, Inc. has set out to define standard interfaces to provide definitions on secure and efficient ways to implement an NTRU system. Although NTRU Cryptosystems is active in numerous bodies, the two most frequently referenced are the Institute of Electrical and Electronics Engineers (IEEE) and the Consortium for Efficient Embedded Security (CEES). Collaboration with the CEES has generated the Efficient Embedded Security Standard #1 (EESS) [13]. Current work with the IEEE has generated the 1363.1 draft standard [14], a work that is still in progress.

To compare the EESS and IEEE 1363.1 documents is inappropriate because the scope differs between the two. The EESS document seeks to provide a standard implementation interface for the NTRU system in wired and wireless applications. The EESS is also targeted more towards applications using a microcontroller and so limits the scope of the parameters recommended. The IEEE 1363.1 document is more of a reference for techniques, theoretic background and security considerations. Due to the difference, the EESS document contains much more material on interface definitions but references the IEEE 1363.1 document for detailed discussion of security. Perhaps due to still being a draft revision, the IEEE 1363.1 document lacks the detailed information on NTRUSign, a signature scheme based on the NTRU operations, which can be found in the EESS document. Despite these differences, the similarities between the two standards are apparent when analyzing the recommended primitives and procedures.

Although the EESS document was taken into consideration during the course of this thesis, the focus was shifted toward the IEEE 1363.1 draft standard because of recent publications. In consideration of future editing to the 1363.1 draft standard, a currently studied attack [15] and the resulting changes caused in the recommended parameter sets [16] was

considered.

2.2.1 IEEE 1363.1

The *IEEE 1363.1 Draft Standard for Public-Key Cryptographic Techniques Based on Hard Problems over Lattices* is part of the IEEE 1363 series of documents meant to provide a central reference for public-key techniques. The 1363-2000 [17] and 1363a-2004 [18] documents provide information for public-key schemes based on the integer factorization, discrete logarithm and elliptic curve discrete logarithm problems. In addition to 1363.1, there is also work being done on *IEEE 1363.2 Password-Based Public-Key Cryptography* [19] and *IEEE 1363.3 Identity-Based Public-Key Cryptography* [20].

2.2.1.1 Supporting algorithms

In order to support the functions of key generation, encryption and decryption certain functionality must be provided. IEEE 1363.1 defines a mask generation function (MGF), index generation function (IGF) and a blinding polynomial generation method (BPGM) which are based on an underlying hash function.

A hash function is used to take a set of bytes (which must not exceed a predefined maximum) and reduce it to a fixed length hash code. A typical use of a hash function is to hash the data being sent to another party and include the hash code with the original message in the encryption process. The party decrypting the message uses the same hash function on the message they decrypt and compares it to the hash code sent with the message to help verify that the correct message has been received. In order for this to be a valid method of verification, the hash function should have a low chance of collisions, two input strings resulting in the same hash code, to obtain the highest chance of detecting any change in the message data during transmission. The currently supported hash functions are the SHA hash functions as defined in FIPS 180-2 [21]. The recommended practice is that for k bits of security, a hash function with k or greater output bits should be used.

A mask generation function is used to ensure a reasonably random distribution of bits

after encryption and to ensure that a single bit of the output is dependent on multiple input bits. From a security standpoint, the encrypted text will require a larger search space to discover and should be less vulnerable to attack if it has been sufficiently randomized. Verification using the MGF output is accomplished during decryption by comparing the MGF output of the recovered message with the MGF value obtained from the encrypted message, the transmission is discarded if the two MGF values differ. Due to one output bit of the MGF being dependent on multiple input bits, this assures with relatively high confidence that the message has not been corrupted during transmission. The MGF function is described in the IEEE 1363.1 document, but can also be found in the IEEE 1363-2000 standard.

In order to provide protection against chosen ciphertext attacks, a blinding polynomial is used during the encryption process whose generation is based on the message being encrypted. An index generation function, similar to the MGF except that it is state aware and can therefore be called multiple times, is used to provide a source of reasonably random indices. To decrease the chance of bias in the output indices from the IGF, the output is constrained to a range that is as close to a multiple of N as possible. The blinding polynomial generation method uses the output indices from the index generation function to create the blinding polynomial. Both the IGF and BPGM are defined in the IEEE 1363.1 document.

2.2.1.2 Optimizations

In addition to the recommended supporting algorithms, there are also recommended practices to efficiently implement the NTRU system. Some of the recommended efficiency improvements are general system optimizations and others are specific to certain choices of parameters.

During the final step of the decryption process, the inverse of the private key modulo the small modulus p is multiplied by the candidate value to obtain the candidate plain text. One of the optimizations used is to choose the form of the private key to be $f = 1 + p * F$,

where F is a random polynomial with dF non-zero coefficients. By choosing the private key in this form, the inverse modulo p of the private key is simply one which eliminates the need for the final convolution multiplication during decryption. Additional savings are achieved in terms of storage and key generation computations, as the inverse of the private key modulo p does not need to be stored or computed.

To obtain greater efficiency during multiplication operations, an alternative form is suggested that takes advantage of sparse polynomials. The suggested form, $f = f_1 * f_2 + f_3$, is constructed from polynomials f_1, f_2, and f_3 having d_{f_1}, d_{f_2} and d_{f_3} non-zero coefficients, respectively. Multiplying these separate vectors, the entire multiplication by f would require $(d_{f_1} + d_{f_2} + d_{f_3})N$ operations instead of $(d_f)N$ operations per coefficient. For example, with $N = 251$ and $d_f = 90$, one might choose $d_{f_1} = d_{f_2} = d_{f_3} = 9$. A multiplication of a polynomial a by f would require $(d_f)N = 22590$ operations per coefficient. Using the alternative representation, the result could be calculated in three steps by $a * f_1$, $(a * f1) * f2$ and $a * f3$, leading to $(d_{f_1} + d_{f_2} + d_{f_3})N = 6777$ operations per coefficient.

Operating under the assumption that construction of message using a binary small modulus would be appealing, an algorithm is provided for efficient multiplication of a binary polynomial with a large modulus reduced polynomial. The algorithm takes advantage of representing the binary polynomial by a vector of positions of the non-zero elements. While such an approach might be applicable to a larger modulus, the efficiency of storing the positions of non-zero elements would be reduced with the need to store what value the non-zero elements were.

2.2.1.3 Security levels and parameter selection

In order to determine the security level of a specific parameter set, work has been done to develop methods to estimate the bit security. For in depth discussion of the estimation procedure for determining the bit security levels used by NTRU, [22] and [23] can be referenced. More recent work has been focused at developing an algorithm to generate a complete parameter set based on the desired final bit security. Description of the algorithm

can be found in [14] and [23]. While the main outlined algorithm seeks to generate as low a value for N as possible to keep the final polynomial sizes small, [23] also presents an alternative approach which fixes the value for q and adjusts N as necessary to obtain the desired security level. The flexibility of this approach allows for implementation of the NTRU not only in larger, performance oriented systems but also in systems which have a hard constraint on the size of integer operands which are allowed.

Chapter 3

NTRU investigation

3.1 Previous work

Although the NTRU algorithm has been scrutinized and peer reviewed by numerous parties, there have not been many widely published implementations in hardware. NTRU Cryptosystems, Inc. has made public some of their own hardware design work [24], while other significant sources include work peformed by Colleen O'Rourke [25] and Jens-Peter Kaps [11] from the Worcester Polytechnic Institute. The most recent source of interest found was a software implementation, performed by Johannes Buchmann et al. at the Technische Universität Darmstadt, that provided a possible increase in efficiency for polynomial multiplication [26].

Work presented in [24] describes the NTRU Embedded Reference Implementation (NERI) toolkit, a multi-platform ANSI C based software package. In addition an FPGA implementation is discussed, although the details are rather limited. More recent packages in C, Java and VHDL work can be purchased at the NTRU website, obviously outside the scope of this work. A significant value provided by the NTRU website is the open disclosure of technical notes and articles.

The work established in [25] demonstrates a scalable architecture to perform NTRU polynomial multiplications and a unified architecture that uses Montgomery multiplication [27] to provide support for NTRU and other cryptographic schemes. Although the architecture presented is relevant in terms of flexibility, better results can now be achieved through

choice of parameters and optimizations that eliminate the need for explicit standard integer multiplication during a polynomial convolution multiplication.

In [11] a scalable ultra low power design is introduced for NTRU polynomial multiplications. The architecture makes use of a circular shift register to rotate the coefficients of one of the polynomial operands during a convolution multiplication. The buffer may be tapped at multiple points to execute concurrent multiplication of coefficients for higher performance or at a single point to reduce power consumption. Due to the focus of the work, it targets a very specific set of conditions in order to make conclusions concerning ultra-low application of NTRU encryption.

A software approach to optimizing the convolution multiplication operation is presented in [26]. The work focuses on the case where a polynomial convolution multiplication occurs between a polynomial with binary coefficients and another general polynomial. By seeking patterns in the binary operand, one calculation could be made to help compute multiple partial results. Although this method was found to provide better results, the choice of using binary polynomials is no longer recommended [16].

3.2 NTRU design and modeling

Although hardware aspects have been considered in previously published works [25] [11] [24], there has been no known publication of an in depth study of the NTRU system regarding implementation in hardware. With the goal of introducing many of the hardware related design issues, the general system will be examined, followed by specific considerations for design and modeling based on the IEEE 1363.1 draft standard. While only the aspects for design are presented here, the following chapter will provide in depth discussion and analysis.

3.2.1 Design considerations for a general NTRU system

For implementation of almost any system, it is often advisable to consider the primary operands and operations that are involved in the system. For the NTRU system, the primary operands are convolution polynomials or their integer coefficients. Operations to consider are addition, multiplication and inversion of convolution polynomials and modular reduction and inversion of coefficients. Although these factors can be evaluated independently of the specific parameters used in the system, often it is more valuable to assess particular groups of parameters.

Depending on the choice of parameters, the length of a polynomial used in the NTRU system can range from around three thousand bits to well into the tens of thousands of bits. Although one of these polynomials may seem to fit trivially in the amount of memory that is commonly available to hardware systems, several polynomials are used in the system and the total amount of memory needed may be unachievable for some systems. In addition, access patterns should be taken into account when deciding on the method of operating on the polynomials, which is often dependent on the type of storage used. At times it may be better to pack coefficients in the minimum space required but there are also situations where it may be better to have padding between coefficients. Some hardware configurations may not support arithmetic on operands above a certain bit length or may be less efficient when operating on bit lengths that are between standard operand boundaries. For example, consider a configurable piece of hardware which has embedded multiplier units accepting 16 bit operands. If the system parameters allow for a minimum coefficients size of 10 bits, then an implementation using only 10 bits might create an implementation using configurable logic or lookup table version of a 10 bit multiplier in order to keep the more efficient embedded resources available. If the option is available, it is possible to force the operation onto the embedded resource anyway, but it would also be possible to pad the coefficients out to 16 bits to cause a migration into the embedded resource. Additional details concerning storage methods and memory requirements can be found in the following chapter.

Following almost every operation in the NTRU system, the result is reduced either modulo p or q. The modular reduction of a polynomial being defined by reducing each coefficient, there are N reductions per operation. Assuming that a general modulus is used, a general reduction algorithm would have to be used, but often the form of the chosen moduli allows for better performance. In hardware, power of two reductions are performed at no cost by truncating the result of an operation or can be performed at minimal cost using a masking operation. Optimizations can also be made to perform faster reduction of certain classes of moduli, such as Mersenne primes which are of the form $p = 2^x - 1$. An example of a fast algorithm for reduction of Mersenne primes is shown in Algorithm 3.1. Modular inversion of integers is used during key generation in the NTRU system

Algorithm 3.1 Modular reduction using Mersenne primes

Input: an integer a, a Mersenne prime p
Output: $b \equiv a \pmod{p}$
Step 1: $b = a$
Step 2: do while $b > p$
Step 3: split b into sections $c_i|c_{i-1}|...|c_1|c_0$ each of length $log_2(p+1)$ bits
Step 4: $b = c_i + c_{i-1} + ... + c_1 + c_0$

depending on the choice of inversion used for polynomials. Polynomial inversion using the Extended Euclidean Algorithm requires an integer modular inversion per iteration and one final inversion to calculate the result. For smaller moduli, the inverse can often be easily stored in a lookup table, but the Extended Euclidean Algorithm can also be used for large moduli. In order to calculate the inverse modulo the power of a prime, an algorithm based on Newton's iteration is presented in [28] and is repeated here, for convenience, in Algorithm 3.2. Although the algorithm is presented for convolution polynomials, the same method is applicable to integers as well.

The convolution polynomial addition operation is the same as normal polynomials in that each coefficient of similar degree is added together. Due to the simplistic nature of the calculation, about the only variable in the process is whether each coefficient is done

Algorithm 3.2 Inversion in $(\mathbf{Z}/p^r\mathbf{Z})[X]/(X^N - 1)$ [28]

`Input:`	$a(X)$, p `(a prime),` r `(the exponent for p)`
	$b(X) \equiv a(X)^{-1} \pmod{p}$
`Output:`	$b(X) \equiv a(X)^{-1} \pmod{p^r}$
`Step 1:`	$q = p$
`Step 2:`	`do while` $q < p^r$
`Step 3:`	$q = q^2$
`Step 4:`	$b(X) := b(X)(2 - a(X)b(X)) \pmod{q}$

individually or whether multiple coefficients of the addition output are calculated simultaneously. The convolution polynomial multiplication operation is far more challenging to efficiently implement and is one of the most researched aspects of the NTRU system. The most obvious manner of calculating the result is to calculate each output coefficient separately by cycling through the N long coefficient vectors. The entire calculation requires iteration through two nested loops and approximately N^2 multiplications and additions. In addition to the methods used in [11] and [26], [29] and [30] present methods for achieving better results for convolution polynomial multiplication. [29] presents a recursive method for splitting the operands of a convolution multiplication until their degree goes below a threshold d. By splitting the operands, partial multiplication results can be reused leading to an overall reduction of the computational work needed to produce the result. The method for improving the efficiency of the multiplication operation presented in [30] is the same optimization presented for [14] which uses operands of a special form. Further analysis of the multiplication operation is provided in the next chapter.

A set of algorithms based on the Extended Euclidean Algorithm is presented in [14] and a set algorithms based on the Almost Inverse Algorithm is presented in [28] for inversion of convolution polynomials. The Extended Euclidean Algorithm is divided into a wrapper function, a call to the Extended Euclidean Algorithm and a polynomial division algorithm in [14]. All of these have been combined to be presented in Algorithm 3.3. The arbitrary

Algorithm 3.3 Extended Euclidean Algorithm - Inversion in $\mathbf{Z}/p\mathbf{Z})[X]/(X^N - 1)$ [14]

Input: A prime p, a positive integer N and a polynomial a in $\mathbf{Z}_p[X]/(X^N - 1)$

Output: A polynomial b satifying $a*b = 1$ in $\mathbf{Z}_p[X]/(X^N - 1)$ if a is invertible in $\mathbf{Z}_p[X]/(X^N - 1)$, otherwise FALSE

Step 1: Set $b := X^N - 1$

Step 2: Set $u := 1$

Step 3: Set $d := a$

Step 4: Set $v_1 := 0$

Step 5: Set $v_3 := b$

Step 6: do while $v_3 \neq 0$

Step 7: Set $t_3 := d$ and $q := 0$

Step 8: Set $u_N := v_{3_N}^{-1} \pmod{p}$

Step 9: do while $deg(t_3) \geq N$

Step 10: Set $dt3 := deg(t_3)$

Step 11: Set $v := u_N * t_{3_{dt3}} * X^{(dt3-N)}$

Step 12: Set $t_3 := t_3 - v * v_3$

Step 13: Set $q := q + v$

Step 14: Set $t_1 := u - q * v_1$

Step 15: Set $u := v_1$

Step 16: Set $d := v_3$

Step 17: Set $v_1 := t_1$

Step 18: Set $v_3 := t_3$

Step 19: If $deg(d) = 0$

Step 20: Return $b = d^{-1} \pmod{p} * u \pmod{X^N - 1}$

Step 21: Else return FALSE

prime version of the Almost Inverse Algorithm provided by [28] is provided in Algorithm 3.4, along with specific versions for $p = 2$ and $p = 3$ as well. For inversion of convolution polynomials modulo the power of a prime, Algorithm 3.2 must be used as an additional step. While it is difficult to immediately choose which is preferable from examination of the algorithms, further analysis of the Extended Euclidean and Almost Inverse Algorithms for convolution polynomial inversion is presented in the next chapter.

Algorithm 3.4 Almost Inverse Algorithm - Inversion in $\mathbf{Z}/p\mathbf{Z})[X]/(X^N - 1)$ [28]

```
Input:    a(X), p  (a prime)
Output:   b(X) ≡ a(X)^{-1}  in  (Z/pZ)[X]/(X^N − 1)
Step 1:   Initialization:  k := 0, b(X) := 1, c(X) := 0,
                           f(X) := a(X), g(X) := X^N − 1
Step 2:   Loop:
Step 3:   do while f_0 = 0
Step 4:       f(X) := f(X)/X, c(X) := c(X) * X, k := k + 1
Step 5:   if deg(f) = 0 then
Step 6:       b(X) := f_0^{-1} b(X) (mod p)
Step 7:       return X^{N−k} b(X) (mod X^N − 1)
Step 8:   if deg(f) < deg(g) then
Step 9:       exchange f and g and exchange b and c
Step 10:  u := f_0 g_0^{-1} (mod p)
Step 11:  f(X) := f(X) − u * g(X) (mod p)
Step 12:  b(X) := b(X) − u * c(X) (mod p)
Step 13:  goto Loop
```

3.2.2 Detailed aspects of IEEE 1363.1

Although many sources provide a basic description of the NTRU system, as is the case with many cryptosystems, additional steps should be taken to protect the system from certain types of attacks. The IEEE 1363.1 draft standard support functions, as described in 2.2.1.1, are provided in detail in this section along with descriptions of the encryption and decryption processes.

To start the discussion of the NTRU system as presented in IEEE 1363.1, an overview of the system parameters is needed. In addition to the basic parameters N, p and q, each sample parameter set also defines certain values in relation to key generation, the formation of the message data, the type of Mask Generation Function (MGF) to use, the type of Blinding Polynomial Generation Method (BPGM) to use, and parameters related to the balancing of the encryption and decryption processes. An example parameter set for $N = 347$, set 2, is provided in Table 3.1.

$N = 347$	(1)
$p = 2$	(2)
$q = 269$	(3)
Key generation: LBP-KGP-1 with	
$dF = 66$	(4)
$dg = 173$	(5)
$lLen = 1$	(6)
$db = 112$	(7)
$dm0 = 108$	(8)
MGF1 with	
SHA-1 (MGF)	
LBP-BPGM1 with	
IGF-MGF1 with SHA-1 (IGF)	
$dr = 66$	(9)
$c = 14$	(10)
$oLenMin = 300$	(11)
$OID = 00\ 02\ 02$	(12)
$pkLen = 0$	(13)
$A = 0$	(14)

Table 3.1: Parameter set for ees347ep2

There are two methods specified for key generation in IEEE 1363.1, one uses the full polynomial forms and the other uses product forms as described in 2.2.1.2. In the case that product forms are not used, as referenced in Table 3.1 and shown in Algorithm 3.5, only the polynomials F and g need to be defined for generation of the private and public keys. Recall that the private key is of the form $f = 1 + p * F$ and the public key is formed by $h \equiv pf_q * g \pmod{q}$. For use in random generation, dF [Table 3.1.4] and dg [Table 3.1.5] define the number of non-zero coefficients that are needed in the polynomials F

and g respectively. In the case that products forms were used, F would be of the form $F = f_1 * f_2 + f_3$ and so a quantity d_{f_i} would be specified for the number of non-zero coefficients in each polynomial f_i.

Algorithm 3.5 Random key generation primitive, LBP-KGP1 (Algorithm 22 in [14])

Algorithm 22 – Random key generation primitive, LBP-KGP1

Components: The domain parameters N, q, p, dF, dg
Input: None
Output: An key pair consisting of the private key f, encoded as a binary ring element of degree N, and the public key h, an element in the ring $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$.
Operation: The key pair shall be computed by the following or an equivalent sequence of steps:

a) Randomly choose a polynomial F of degree $N - 1$ with dF coefficients equal to 1 and the remaining coefficients equal to 0.

b) Compute the polynomial $f := 1 + p*F$ in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$

c) Compute the polynomial f^{-1} (i.e. the polynomial f^{-1} such that $f^{-1}*f = f* f^{-1} = 1$) in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$. If f^{-1} does not exist, go to step 1.

d) Randomly choose a polynomial g of degree $N - 1$ with dg coefficients equal to 1 and the remaining coefficients equal to 0.

e) Check that g is invertible mod q. If it is not, go back to step 4.

f) Compute the polynomial $h := f^{-1}*g*p$ in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$

g) Output f, h

Conformance region recommendation: A conformance region shall include a valid set of components (N, q, p, dF, dg).

The first part (steps a through f shown in Figure 3.6) of the encryption process is to form the message construct $M = b \parallel octL \parallel m \parallel p0$. The message data m is an input to the encryption function along with l, the length of the message m in bytes. The $lLen$ [Table 3.1.6] parameter defines the number of bytes that should be used to encode the value of l, which is stored into $octL$ for use in constructing M. The quantity db [Table 3.1.7] describes the number of bits that should be in the randomly generated bit string b used in M. The last part of the message construct, $p0$, is a zero pad used to pad M out to $nLen = \lceil N/8 \rceil$ bytes, the number of bytes needed to hold N bits.

The second part (steps g through o shown in Algorithm 3.6) of the encryption process forms the final data used in encryption, an XORing of the message construct M and a blinding value. Similar to key generation, there are two methods specified for generation of the blinding value depending on whether product forms are used or not. In Table 3.1, LBP-BPGM1 refers to the blinding polynomial generation method (BPGM) which does not use product forms. Furthermore, IGF-MGF1 is defined as the underlying index generation function (IGF) for the BPGM and SHA-1 is defined as the underlying hash function for the IGF. For reference, the functional diagram representing the flow of variables used during the encryption process is provided in Figure 3.1.

Algorithm 3.6 Encryption Operation (Algorithm 26 in [14])

Algorithm 26 – Encryption operation

Components:

— The parameters N, q.
— The length of the encoded length $lLen$.
— The number of bits of random data db, which must be a multiple of 8.
— The chosen Mask Generation Function and Hash Function.
— The chosen Blinding Polynomial Generation Method and the associated parameters
— The OID, an octet string
— The number of bits of public key to hash, $pkLen$.
— The minimum message representative weight, $dm0$.

Inputs:

— The message m, which is an octet string of length l octets
— The public key h, which is a (q, N)-ring element.

Output:

— The ciphertext e, which is a (q, N)-ring element, or the errors "invalid key" or "message too long"

Operation: The ciphertext e shall be calculated by the following or an equivalent sequence of steps:

a) If desired, validate or perform plausibility tests on the public key h. If the key fails the tests, output "invalid key" and exit.

b) Calculate:

1) $nLen$ = ceil $[N/8]$, the number of octets required to hold N bits.

2) $octL$ = the $lLen$-octet-long encoding of the message length l.

3) $bLen = db/8$, the length in octets of the random data.

4) $maxLen = nLen - 1 - lLen - bLen$, the maximum message length.

c) If $l > maxLen$, output "message too long" and stop.

d) Randomly select an octet string b of length $bLen$.

e) Form the octet string $p0$, consisting of the 0 byte repeated $(maxLen + 1 - l)$ times.

f) Form the octet string M of length $nLen$ as $b \| octL \| m \| p0$.

Algorithm 3.7 Encryption Operation - continued (Algorithm 26 in [14])

Algorithm 26 – Encryption operation

g) Convert the public key h to a bit string bh using RE2BSP. Form the bit string $bhTrunc$ by taking the first $pkLen$ bits of bh. Convert $bhTrunc$ to the octet string $hTrunc$, of length $pkLen/8$ using BS2OSP. Form $sData$ as the octet string OID $\| m \| b \| hTrunc$

h) Use the chosen blinding polynomial generation method with the seed $sData$ and the chosen parameters to produce r.

i) Calculate $R = r*h \bmod q$.

j) Calculate $R2 = R \bmod 2$.

k) Convert $R2$ to the octet string $oR2$ using BRE2OSP.

l) Form m' by putting $oR2$ through the chosen MGF/Hash and XORing the leading $nLen$ bytes of the output with M.

m) Set the leading $((nLen * 8) - N)$ bits of the final octet of m' to 0.

n) If $dm0 > m'(1)$ or $m'(1) > N\text{-}dm0$, return to step d).

o) Convert m' to i, a binary polynomial of length N, using OS2BREP.

p) Calculate the ciphertext as $e = R + i \bmod q$.

q) Output e.

Conformance region recommendation: A conformance region shall include:

— at least one set of components as defined above

— at least one valid public key h consistent with those components. Ff key validation is performed, invalid public keys that are appropriately rejected by the implementation may also be included in the conformance region

— all messages of length less than or equal to $maxLen$ for a given set of components.

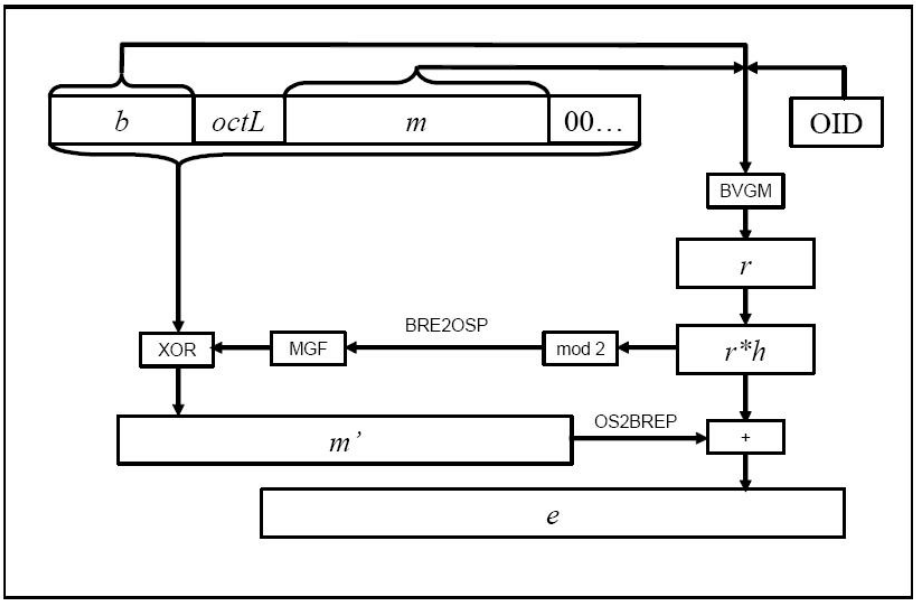

Figure 3.1: Functional diagram of the encryption operation [14]

The index generation function (IGF), shown in Algorithm 3.8, uses an input seed and a counter to run a hash algorithm multiple times in order to generate pseudo-random index data. The index data is used by the blinding polynomial generation method (BPGM), shown in Algorithm 3.10, to create the non-zero coefficients of the blinding polynomial r. The seed for the IGF, $sData$ = OID $\|$ m $\|$ b $\|$ $hTrunc$, is constructed from the message data m, the random bit padding b, an identifier for the parameter set OID [Table 3.1.12] and $hTrunc$, the first $pkLen$ [Table 3.1.13] bits of the public key. In order to prevent timing based attacks based on the number of calls to the hash function during the generation of the blinding polynomial, as described in [31], the parameter $oLenMin$ is provided as the minimum number of output bytes that should be generated by the IGF. The IGF takes as an input the minimum number of calls that should be made to the underlying hash function, minCalls, which can be calculated use the $oLenMin$ [Table 3.1.11] parameter and the number of bytes of output that the underlying hash generates, $hLen$, by minCalls $= oLenMin/hLen$. In the parameter set shown in Table 3.1, the value of $hLen$ would be 20 and so minCalls $= 15$. The IGF consumes the stored buffer of hash

values $oLen$ bytes at a time, however $oLen$ is not defined in the parameter sets but can be experimentally determined if sample data is provided. The indices output from the IGF are taken from the leading $NLen = \lceil (\log_2 N)/8 \rceil$ bytes of the $oLen$ bytes consumed during each call. To restrict the range of the index generation output, only the trailing c [Table 3.1.10] bits of the $NLen$ bytes are used. The BPGM creates a polynomial with all zero coefficients to start with, repeatedly calling the IGF to receive an index to check in the polynomial. If the coefficient is non-zero, then the index is discarded and the IGF is called again, but otherwise the coefficient is set and the number of non-zero coefficients left to assign is decreased by one. The end result is the blinding polynomial r, which will have dr [Table 3.1.9] non-zero coefficients.

Algorithm 3.8 Index generation function (Algorithm 19 in [14])

Algorithm 19 – Index generation function (IGF-MGF1)

Components: A hash function *Hash* with output length *hLen* octets.
Input:
EITHER: an octet string Z of length *zLen* octets; the modulus N, an integer; the index generation constant c, an integer; and the minimum number of calls minCalls, an integer;
OR: the state s.
Output: An octet string o of length *oLen* octets and the state s; or "error".
Operation: The octet string o shall be produced by the following or an equivalent sequence of steps:

a) If s is not provided:

1) If *zLen*+4 exceeds any input length limitation on the hash function *Hash*, output "error" and exit

2) If *minCalls* exceeds 2^{32}, output "error" and exit.

3) Initialize *totLen* to 0. Intialize *remLen* to 0.

4) Initialize the octet string *buf* to be a zero-length octet string.

5) Initialize *counter*:= 0.

6) Initialize N and c with the provided values. Set *NLen* = ceil ($\log_2(N)/8$).

7) While *counter* < *minCalls* do

i) Convert *counter* to an octet string C of length 4 octets using I2OSP.

ii) Compute *Hash*(Z || C) with the selected hash function to produce an octet string H of length *hLen* octets.

iii) Let *buf* = *buf* || H.

iv) Increment *counter* by one.

8) Set *remLen* = *totLen* = *minCalls* * *hLen*.

Algorithm 3.9 Index generation function - continued (Algorithm 19 in [14])

Algorithm 19 – Index generation function (IGF-MGF1)

b) Otherwise (if *s* is provided):

   1) Extract the values *totLen*, *remLen*, *buf*, *counter*, *N*, *NLen* and *c* from the state *s*. (The details of how they are stored in *s* may be determined by the implementer).

c) Set *totLen*:=*totLen* + *NLen*.

d) If *totLen* exceeds $hLen \times 2^{32}$, output "error" and exit.

e) If *remLen* < *NLen*

   1) Let the octet string *M* be the trailing *remLen* octets in *buf*.

   2) Let *tmpLen*:=*NLen* – *remLen*.

   3) Let *cThreshold* = *counter* + *ceil*[*tmpLen*/*hLen*].

   4) While *counter* < *cThreshold* do

      i) Convert *counter* to an octet string *C* of length 4 octets using I2OSP.

      ii) Compute *Hash*(*Z* || *C*) with the selected hash function to produce an octet string *H* of length *hLen* octets.

      iii) Let *M* = *M* || *H*.

      iv) Increment *counter* by one. If *tmpLen* > *hLen*, decrement *tmpLen* by *hLen*.

   5) Set *remLen*:=*hLen* – *tmpLen*. Set *buf*:=*H*.

f) else

   1) Set *M* equal to the trailing *remLen* octets of *buf*.

   2) Set *remLen*:=*remLen* – *oLen*.

g) Set the octet string *o* to the leading *NLen* octets in *buf*.

h) Set the leftmost 8*NLen* – *c* bits of *o* to 0.

i) Convert *o* to an integer *i* using OS2IP.

j) If ($i > 2^c - (2^c \bmod N)$) go back to step c.

k) Store the values *totLen*, *remLen*, *buf*, *counter*, *N*, *NLen* and *c* in the state *s*. (The details of how they are stored in *s* may be determined by the implementer).

l) Output *i* and *s*.

Algorithm 3.10 Blinding polynomial generation method (Algorithm 20 in [14])

Algorithm 20 – Blinding polynomial generation from *dr*, LBP-BPGM1

Components: The parameters N and dr, the chosen index generation function IGF( ), the hash function Hash() chosen to parameterize IGF( ), the polynomial index generation constant c, and the minimum number of hash calls for the IGF to make, $minCalls$.
Input: The seed, which is an octet string *seed*.
Output: The blinding polynomial, which is a polynomial r.
Operation: The blinding polynomial shall be computed by the following or an equivalent sequence of steps:

a) Instantiate the index generation function with hash function Hash() and input *seed*, N, c, $minCalls$ to obtain the IGF state s.

b) Set $t := 0$

c) Set $r := 0$

d) While $t < dr$ do

1) Call the IGF to obtain an integer i.

2) If $r_{(i \bmod N)} = 0$

i) Set $r_{(i \bmod N)} := 1$

ii) Set $t := t + 1$

e) Return r

Once the blinding polynomial, r, has been generated, it is multiplied by the public key to form $R = r * h \pmod q$. To calculate the mask, R is taken modulo two and input into the mask generation function (MGF). The MGF, shown in Algorithm 3.11, calculates the hash of the input concatenated with a 32-bit counter for $cThreshold = \lceil oLen/hLen \rceil$ iterations to fill a buffer of size $cThreshold * hLen$ bytes, where $oLen$ can be considered to be $nLen = \lceil N/8 \rceil$. The result of the MGF is XOR'ed with the message contruct M to form the message representative, m'. After zeroing out the difference between N and the number of bytes that N fits into, $nLen$, the message representative is checked to see that it has between $dm0$ [Table 3.1.8] and $N - dm0$ non-zero bits to ensure that it is decently balanced in the number of zero and non-zero bits. The final encrypted value, $e \equiv R + i \pmod q$, is calculated by converting the message representative into a polynomial, i, and

adding it to the value of the blinding polynomial multiplied by the public key.

Algorithm 3.11 Mask generation function (Algorithm 18 in [14])

Algorithm 18 – Mask Generation Function (MGF1)

Components: A hash function *Hash* with output length *hLen* octets.
Input: An octet string *Z* of length *zLen* octets, and the desired length of the output, which is a positive integer *oLen*. (*oLen* shall be less than or equal to $hLen \times 2^{32}$).
Output: An octet string *mask* of length *oLen* octets; or "error".
Operation: The octet string *mask* shall be produced by the following or an equivalent sequence of steps:

a) If *oLen* exceeds $hLen \times 2^{32}$, or if *zLen*+4 exceeds any input length limitation on the hash function *Hash*, output "error" and exit.

b) Let *M* be the empty string. Let *cThreshold* = ceil[*oLen/hLen*].

c) Set *counter* := 0.

d) While *counter* < *cThreshold* do

   1) Convert *counter* to an octet string *C* of length 4 octets using I2OSP.

   2) Compute *Hash*(*Z* || *C*) with the selected hash function to produce an octet string *H* of length *hLen* octets.

   3) Let *M* = *M* || *H*.

   4) Increment *counter* by one.

e) Output the leading *oLen* octets of *M* as the octet string *mask*.

The decryption operation, shown in Algorithm 3.12, reverses the encryption operation using the decryption primitive, shown in Figure 3.14. The decryption primitive uses the parameter A [Table 3.1.14] to decrypt the coefficients of the message representative in the proper range and is called by the decryption operation algorithm. The decryption operation retrieves the candidate message representative, ci, from the decryption primitive and uses it to calculate the candidate value for the product of the blinding value and the public key, $cR = e - ci$. The candidate message construct, cM, is retrieved by running cR through the MGF function and XORing the result with ci (XORing a quantity by the same value twice is equivalent to XORing the quantity by all zeroes, which does not change the data).

The candidate bit padding, cb, and candidate message data are used along with the OID to generate a candidate blinding polynomial cr. The candidate blinding value is multiplied by the public key, to form cR', which is then checked against cR. If the two values match, then the decryption operation is successful. For reference, the functional diagram representing the flow of variables used during the decryption process is provided in Figure 3.2.

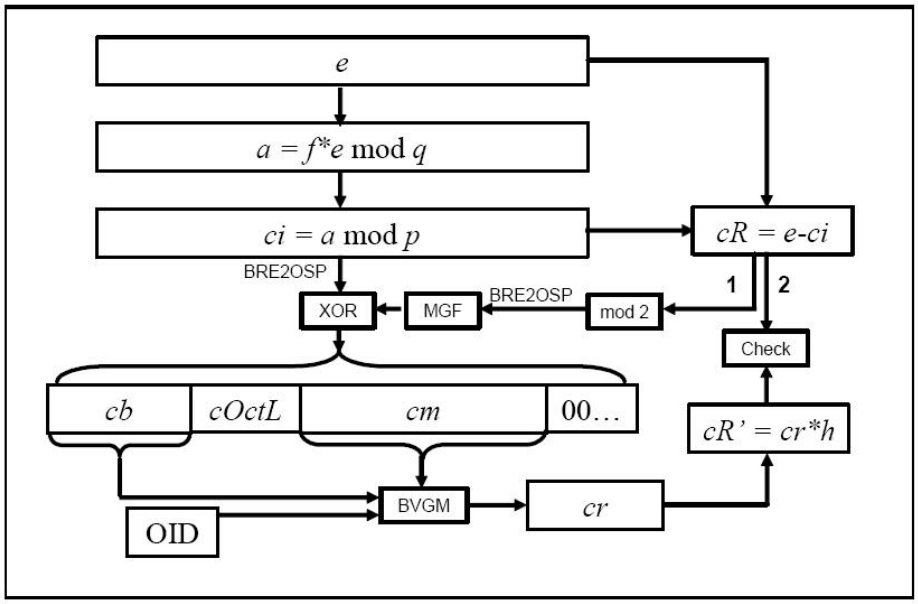

Figure 3.2: Functional diagram of the decryption operation

3.2.3 VHDL modeling of an IEEE 1363.1 system

To investigate the aspects of the NTRU system, a hybrid behavioral and structural VHDL model was designed. Components that were easily translatable to hardware were implemented using structural models, while some of the more complex components were written using behavioral style code. Two main goals were used during the design and creation of the IEEE 1363.1 system model. The first goal was to make the system as modular as possible to facilitate the changing of individual models without need for recreation of the full system model. Due to the draft status of the IEEE 1363.1 standard, it is likely that future edits could change the recommended practices, which is reflected in the separation

Algorithm 3.12 Decryption operation (Algorithm 27 in [14])

Algorithm 27 – Decryption operation

Components:
— The domain parameters N, q, p
— The LBP-PKE decryption primitive to use
— The length of the encoded length $lLen$.
— The number of bits of random data db, which must be a multiple of 8.
— The chosen Mask Generation Function and Hash Function.
— The chosen Blinding Polynomial Generation Method and the associated parameters
— The OID, an octet string
— The number of bits of public key to hash, $pkLen$.
— The lower bound A
— The minimum message representative weight $dm0$

Inputs:
— The ciphertext e, which is a (q, N)-ring element.
— The private key f or (f, f_p).
— The public key h

Output:
— The message m, which is an octet string, or "fail".

Operation: The message m shall be calculated by the following or an equivalent sequence of steps:

a) Set $fail = 0$.

b) Calculate:

1) $nLen$ = ceil $[N/8]$, the number of octets required to hold N bits.

2) $bLen = db/8$, the length in octets of the random data

3) $maxLen = nLen - 1 - lLen - bLen$, the maximum message length.

c) Decrypt the ciphertext e using the selected NTRU decryption primitive with components N, q, p, A and inputs e and f to get the candidate decrypted polynomial ci.

d) Calculate the candidate value for $r*h$, $cR = e - ci$.

e) Calculate $cR2 = cR \bmod 2$.

f) Convert $cR2$ to the octet string $coR2$ using BRE2OSP.

g) Convert the binary polynomial ci to the octet string cm' using BRE2OSP.

h) If $cm'(1) > N\text{-}dm0$ or $cm'(1) < dm0$, set $fail = 1$.

i) Form the candidate padded message cM by putting $coR2$ through the chosen MGF/Hash and XORing the leading $nLen$ bytes of the output with cm'.

j) Set the leading $((nLen * 8) - N)$ bits of the final octet of cM to 0.

Algorithm 3.13 Decryption operation - continued (Algorithm 27 in [14])

Algorithm 27 – Decryption operation

k) Parse *cM* as follows.

1) The first *bLen* octets are the octet string *cb*.

2) The next *lLen* octets are *cOctL*, the message length encoded as an octet string. Convert *cOctL* to the candidate message length *cl* using OS2IP. If $cl > maxLen$, set $fail = 1$ and set $cl = maxL$.

3) The next *cl* octets are the candidate message *cm*.

4) If they are not, set $fail = 1$.

l) Convert the public key *h* to a bit string *bh* using RE2BSP. Form the bit string *bhTrunc* by taking the first *pkLen* bits of *bh*. Convert *bhTrunc* to the octet string *hTrunc*, of length *pkLen*/8 using BS2OSP. Form *sData* as the octet string

OID || *cm* || *cb* || *hTrunc*

m) Use the chosen blinding polynomial generation method with the seed *sData* and the chosen parameters to produce *cr*.

n) Calculate $cR' = h * cr \bmod q$.

o) If $cR' \mathrel{!}= cR$, set $fail = 1$

p) If $fail = 1$, output "fail". Otherwise, output *cm* as the decrypted message *m*.

Conformance region recommendation: A conformance region shall include:

— at least one set of components as defined above

— at least one valid private key *f* and the corresponding public key *h*, consistent with the components.

— for each set of components, all ciphertexts consistent with the components, i.e. all *e* such that *e* is a (q, N)-ring element.

Algorithm 3.14 Decryption primitive (Algorithm 24 in [14])

Algorithm 24 – Decryption primitive, LBP-DP1

Components: The parameters N, q, p, the lower-bound decryption coefficient A
Input: The recipient's private key f, and the encrypted message representative, which is a ring element e in the ring defined by (N, q).
Output: The message representative candidate, which is a binary ring element i' of dimension N.
Operation: The message representative candidate shall be computed by the following or an equivalent sequence of steps:

a) Compute the product $a := f*e$ in $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$ with coefficients reduced into the interval $[A, A + q - 1]$.

b) Calculate the mod p reduction of the product computed in step 1 to obtain the message representative candidate i'.

Conformance region recommendation: A conformance region should include:
— at least one valid set of components (N, q, p, A)
— at least one valid private key f
— all purported ciphertexts that can be input to the implementation; this consists of all e such that e is an element in the ring $(\mathbf{Z}/q\mathbf{Z})[X]/(X^N - 1)$

of the system into fairly small functional blocks. Each of the functional blocks making a sensible division, the models are mainly separated around the boundaries of the algorithms presented in the standard, as shown in Figure 3.3.

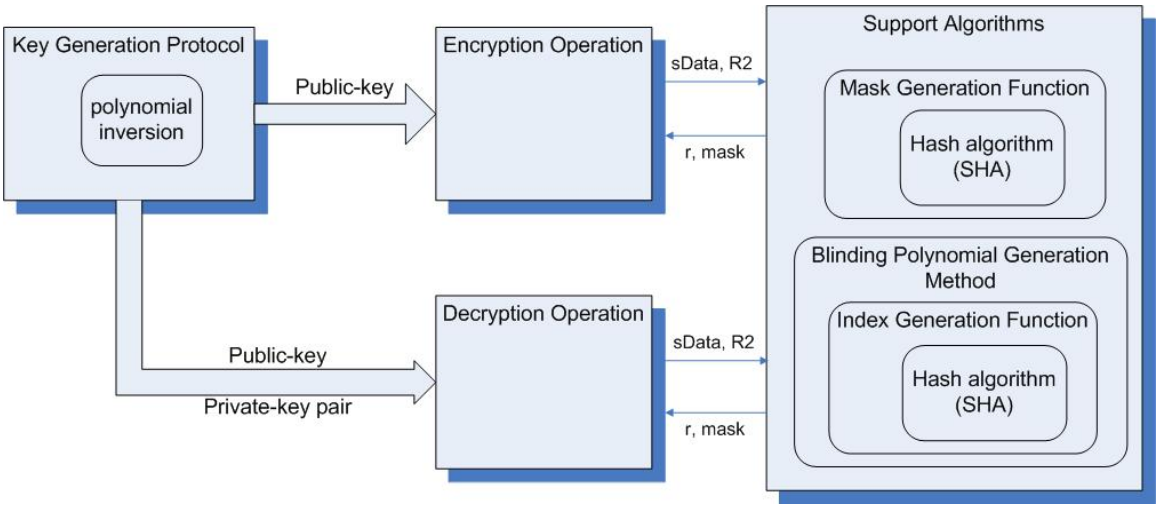

Figure 3.3: Interaction of IEEE 1363.1 system components

The second goal was to make the system flexible with regard to changing of the top-level system parameters. Although the system could have been created in a manner similar to the algorithm blocks defined in the IEEE 1363.1 draft standard, the choice was instead made to use generic declarations to control each model. Use of generics in each model allowed the system testbench to control the parameters used in testing without modification of each individual model but also allowed for a reduced number of inputs and outputs compared to that needed for a dynamically changing system. The full models for key generation, encryption and decryption were wrapped in a testbench which provided inputs based on available testing data and checked for the expected outputs. The final model became a testing platform for all of the pieces of the IEEE 1363.1 draft standard with ability to be piece-wise adapted for further study and optimization. Figure 3.4 shows the IEEE 1363.1 components with reference to the VHDL models created from them and the testing loop used to verify the final incorporated models.

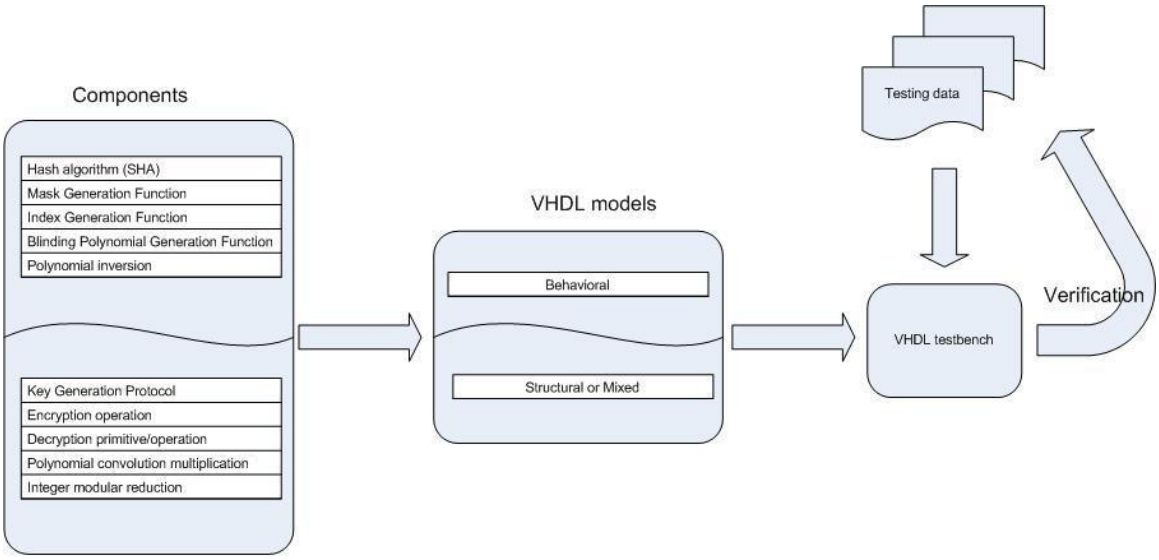

Figure 3.4: Design flow for creation and testing of VHDL models

In order to facilitate the convolution multiplications of a general system, a model was created to handle all of the cases that might need to be handled. The model, shown in Figure 3.5, accepts two polynomials modulo p, two polynomials modulo q or a combination of

the both. In addition, the output polynomial can be reduced modulo an integer input, *modn*, to allow for special cases and also allows a scaling of the output polynomial, through use of the integer input *scale*, before reduction for calculations such as the public key where the output needs to be scaled by p. The type of input used and whether or not the output is scaled is controlled by the input *datyp*. The calculation that performs the convolution multiplication is the basic algorithm, taking N^2 operations, where each operation is a multiplication and an addition. The calculation is initiated by a rising edge on the *daclk* and is calculated immediately due to the behavioral nature of the model. In addition to the modulo p and modulo q outputs, there is a modulo two output used for directly inputting into the mask generation function after the calculation of $R = r * h \pmod{q}$. The generic values *cl2q* and *cl2p* are used throughout the system and represent the number of bits that should be used to store each coefficient in q and p respectively.

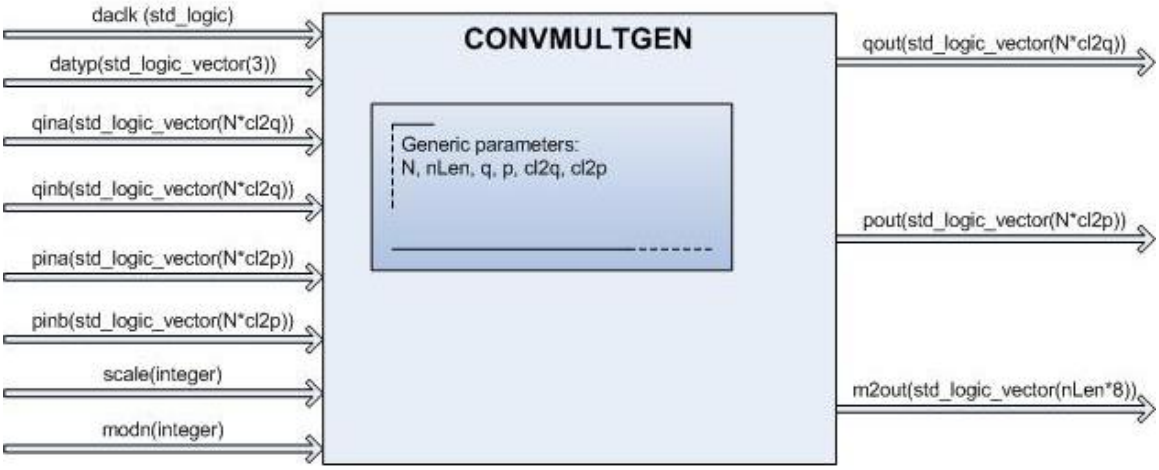

Figure 3.5: Block diagram for basic, general convolution multiplication

The underlying building block for the mask generation function and index generation function is a hash function. Under the current draft standard of IEEE 1363.1, the Secure Hash Algorithm (SHA) family of hashes are used as specified in [21]. Due to the volume of research that has been conducted on the SHA family of hashes, a minimum level of

effort was placed in generating versions of the specific SHA hashes used in testing. A VHDL procedure was constructed for each version of SHA needed, an example of which is shown in Figure 3.6. The diagram shows the internal usage of the procedure to calculate the hash along with a wrapper model to allow for control of the inputs and output of the hash, although this is merged with the inputs and outputs of the functional blocks that the procedure is included in. Upon a rising edge transition on the *daclk* input, the current 512-bit block of the input message of *msize* is processed and output to *hashv*. Subsequent rising edges on the *daclk* will continue to cause processing on the *mblock* input using the previous hash values until either a reset occurs or the full number of bits in the message have been processed. Although the choice of using a procedure caused the hash implementation to generate one MGF and one IGF model for each SHA version, the impact was minimal and the model could be easily adapted to a new hash procedure or modified to calculate the hash in a separate model. The decision on which hash function to used is determined by the security level, k, desired. The recommendation is to choose a hash function with output length greater than or equal to k, although the recommended parameters for security level $k = 160$ use SHA-256 instead of SHA-1. Although this research was conducted using the recommended SHA functions, it should be noted that the National Institute of Standards and Technology (NIST) is currently accepting submissions for a new hash algorithm to augment the FIPS 180-2 Secure Hash Standard. The new algorithm, dubbed "SHA-3", will seek to provide better security compared to the previous SHA functions, in light of advancements in the cryptanalysis of hash functions [32].

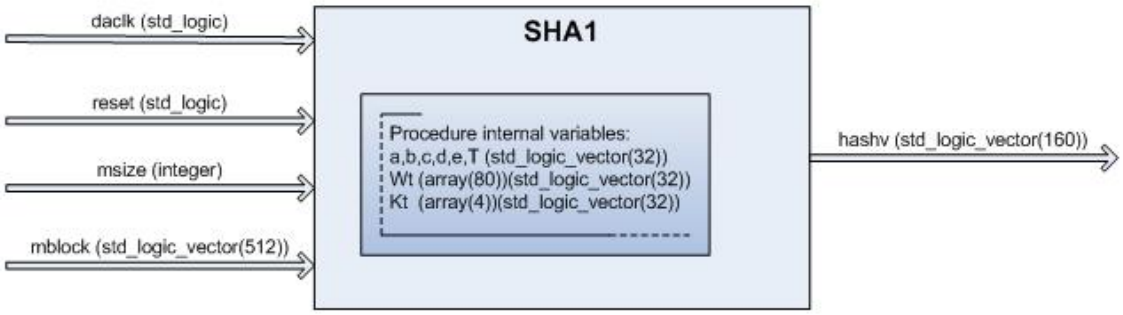

Figure 3.6: Block diagram for SHA1

The mask generation function (MGF) model, shown in Figure 3.7, is mainly a wrapper for either the SHA-1 or SHA-256 procedure, so that the hash can be called multiple times. Since the input to the MGF is known to be $R2 \equiv r * h \pmod 2$, the *zblock* input is known to be $nLen = \lceil N/8 \rceil$ bytes. The number of input blocks, *numblk*, is calculated to be the number of hash input block needed to fit $nLen + 32$ bits, where the additional 32 bits are needed for a counter. If the output of the hash function is long enough to produce $nLen$ bytes of output, then only one iteration is executed, otherwise the counter is incremented and the hash function is called again. The hash output of each consecutive iteration is concatenated together until enough data has been buffered that it can be XOR'ed with the entire message construct. The value of *oLen* is the number of hash output blocks that are needed for the masking operation.

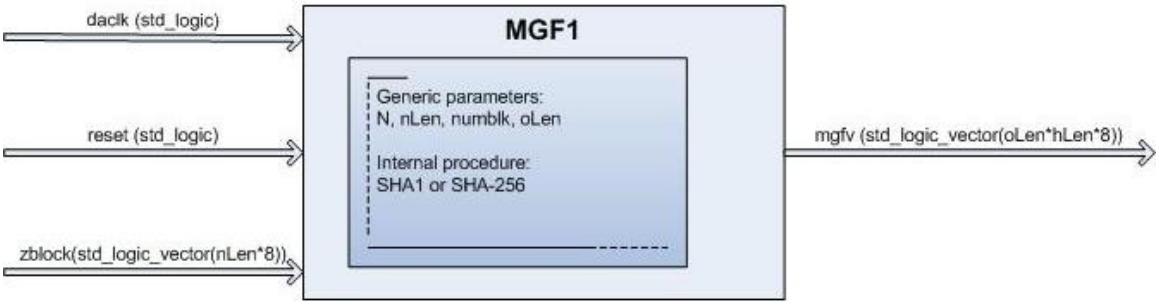

Figure 3.7: Block diagram for MGF1

The index generation function model, shown in Figure 3.8, also wraps around either

the SHA-1 or SHA-256 procedure, but adds extra functionality to maintain status for subsequent calls and to control the output integer value on each call. The input *zblock* is the seed data $sData = \text{OID} \parallel m \parallel b \parallel hTrunc$, which is of a known maximum length. Since all of the sample parameter sets use $pkLen = 0$, zero bytes of the public key are used in the seed data, so it was not considered in the calculation for the maximum length of the seed data. Although the input *zblock* is expanded based on the maximum message size, the input *octL* is used during the calculation of the hash to determine how many bytes of message data are actually present. The parameter *numblk* is used in a similar fashion to the mask generation function, it is used to create a buffer that will fit the maximum length input data for input into the hash algorithm. Due to a conflict with the variable named used in SHA, the system parameter c is redefined as *IGFc* in the index generation function.

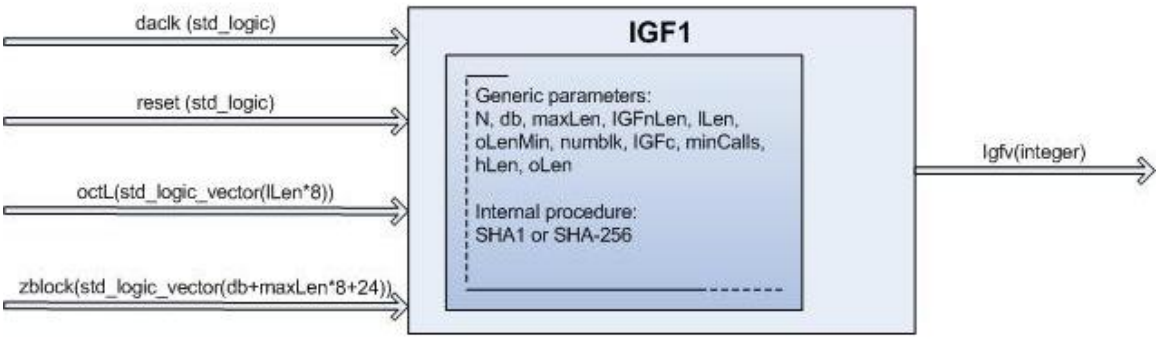

Figure 3.8: Block diagram for IGF1

The blinding polynomial generation method (BPGM) model, shown in Figure 3.9, is implemented as a simple state machine used to generate the blinding polynomial r. The state machine enters an active state upon a high input on the *strtgen*, at which point it clocks the *daclk* input to the index generation function to receive a pseudo-random index. The index received is checked in the polynomial r and set if the coefficient is not already non-zero. If a coefficient can successfully set, the remaining count of coefficients that need to be set is decremented by one until the full dr coefficients have been assigned. Once the full number of coefficients have been assigned, the *gendone* signal is set high and the output

polynomial data is held valid until the *strtgen* signal is set low. If the *strtgen* signal is set low before the state machine reaches the output holding state, then the output polynomial will be held valid for only one clock cycle.

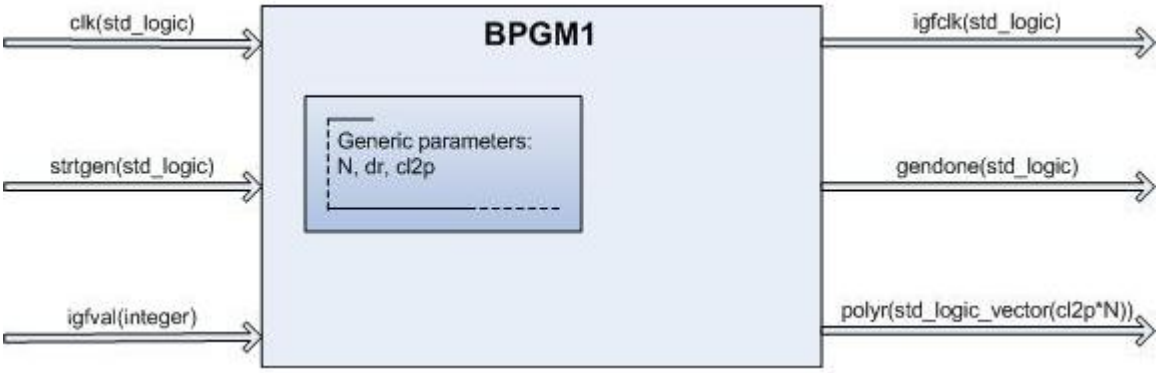

Figure 3.9: Block diagram for BPGM1

The key generation primitive (KGP) model, shown in Figure 3.10, interfaces with the convolution multiplication model and the inverse polynomial model to generate the private and public keys. The KGP is implemented as a state machine which is activated and deactivated in a similar manner to the blinding polynomial generation method state machine. The KGP accepts as input the random polynomials F and g, which were obtained from available test data. The state machine starts by calculating the private, $f = 1 + p * F$. The inverse polynomial model is then used and the output is checked to make sure that the inverse of the private key exists modulo q. If the inverse does not exist, the state machine returns to loading the input value for the polynomial F and starts the calculation of the private key and its inverse again. If the inverse of the private key does exist, the state machine loads the value of the random polynomial g and calculates the inverse modulo q. Although it is not necessary for g to be invertible, the calculation of security used in the IEEE 1363.1 draft standard requires that the public key be invertible modulo q, which in turn requires each of its components to be invertible modulo q. In the same manner as with F, if the inverse of g does not exist, the state machine reloads the value of g and recalculates the inverse. The consequence of implementing the KGP this way is that the testbench which

operates the KGP must respond to the invalid inverse signals to ensure that new random polynomials are input in case of an inversion failure, otherwise the state machine will enter an infinite loop. After validating that the private key and g have valid inverses modulo q, the inverse of the private key and the polynomial g are used as inputs to the convolution multiplication model along with p as the input scale quantity so that the calculation of the public key, $h = pf_q * g \pmod q$, is accomplished all at the same time. The private key, f, and public key, h, are then held valid on the output until the *strtgen* is driven low.

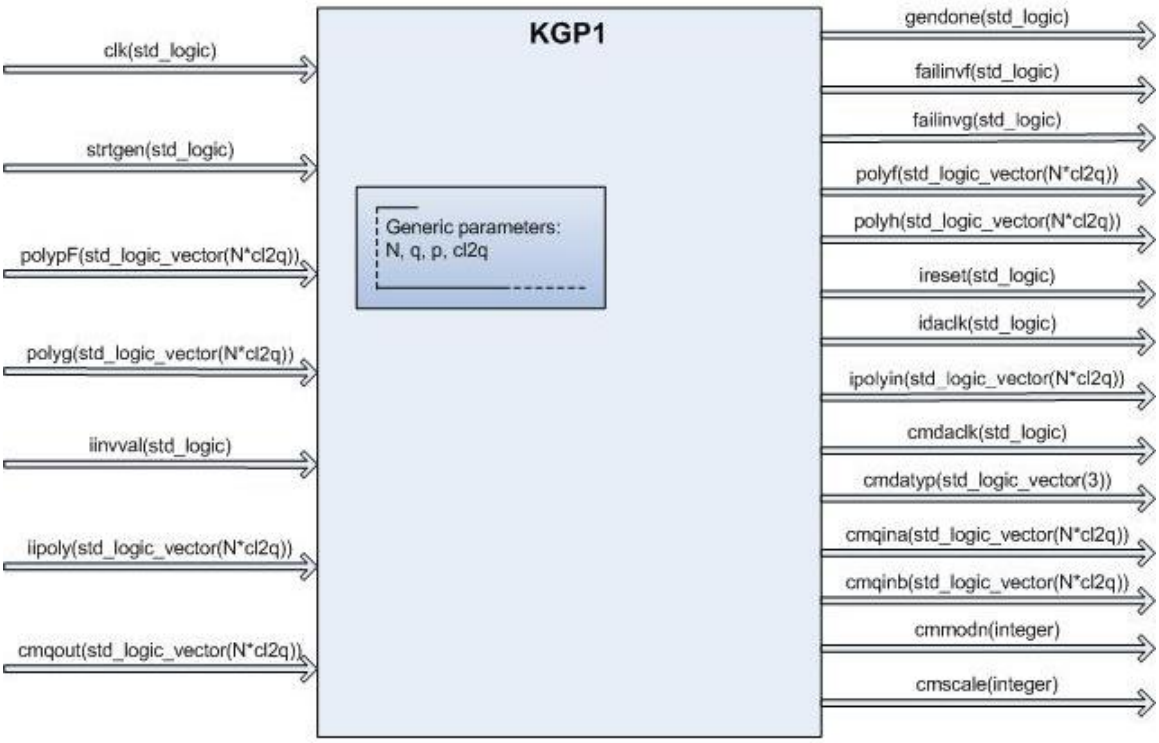

Figure 3.10: Block diagram for KGP1

The polynomial inversion modulo q model, shown in Figure 3.11, combines the algorithms for polynomial division, Extended Euclidean Algorithm and inversion of a polynomial modulo a prime as shown in Algorithm 3.3. The model takes a polynomial modulo q as an input and is activated by a rising edge on the *daclk* input. The implementation is behavioral, so the algorithm runs during one clock cycle and outputs the inverse of the input polynomial and whether the inverse is valid. The inverse polynomial is held on the

output and state is maintained, so the model should be reset to clear all stored values before another inverse polynomial calculation is executed.

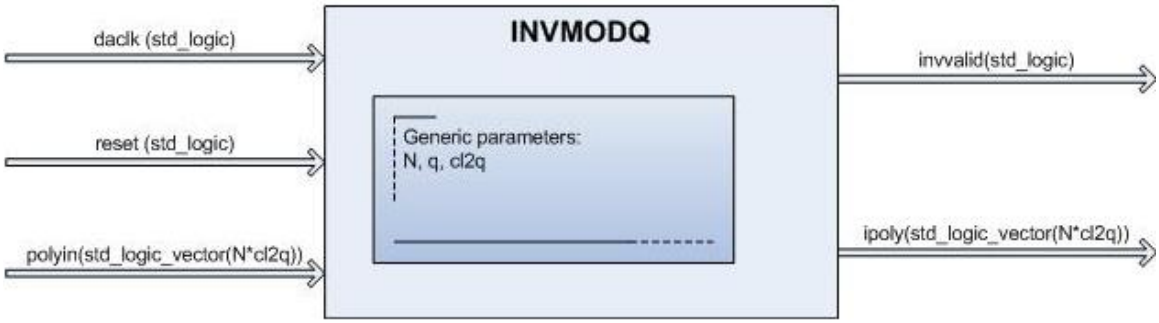

Figure 3.11: Block diagram for inversion mod q

During testing, the encryption operation model, shown in Figure 3.12, retrieves the public key from the key generation primitive via the testbench. The testbench also provides the input message, from testing data, to the encryption operation and drives the *strtenc* signal to enable encryption. The message construct, M, and index generation function seed data, $sData$, are formed and the seed data is sent directly to the index generation function. The blinding polynomial generation method model is then activated, after which the output blinding polynomial, r, is multiplied by the public key, h, using the convolution multiplication model. The output modulo q, R, and modulo two, $R2$, are received from the convolution multiplication model and $R2$ is run through the mask generation function and masked with the message construct. Finally, the encryption operation forms the ciphertext, $e = R + i \pmod q$. An additional parameter, *zeromsk*, is pre-generated so that the mask used in step m of the encryption operation, as shown in Figure 3.6, did not have to be dynamically created for each parameter set.

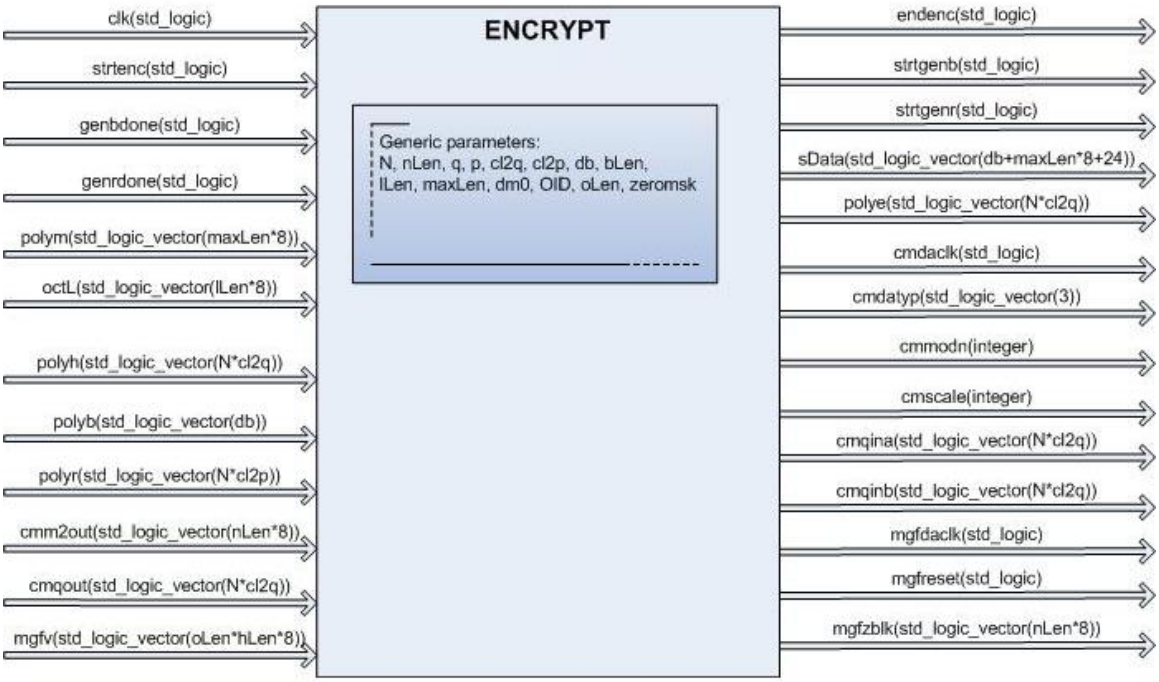

Figure 3.12: Block diagram for encryption operation

The decryption operation model, shown in Figure 3.13, is set up in the testbench to accept the ciphertext, e, from the encryption operation model and the private key, f, from the key generation primitive model. The decryption model uses internal states to execute the decryption primitive to recover the candidate decrypted polynomial, ci. The candidate value for $r * h$ is recovered by $cR = e - ci$ and taken modulo two to be used in the mask generation function. The resultant mask is XOR'ed with ci and the candidate message construct, cM, is retrieved and held as an output from the decryption operation model. Although the candidate message construct was verified manually using the testing data, it should be noted that the blinding polynomial generation method model could easily be integrated with the decryption operation model for automated verification of the data as shown in Figure 3.2.

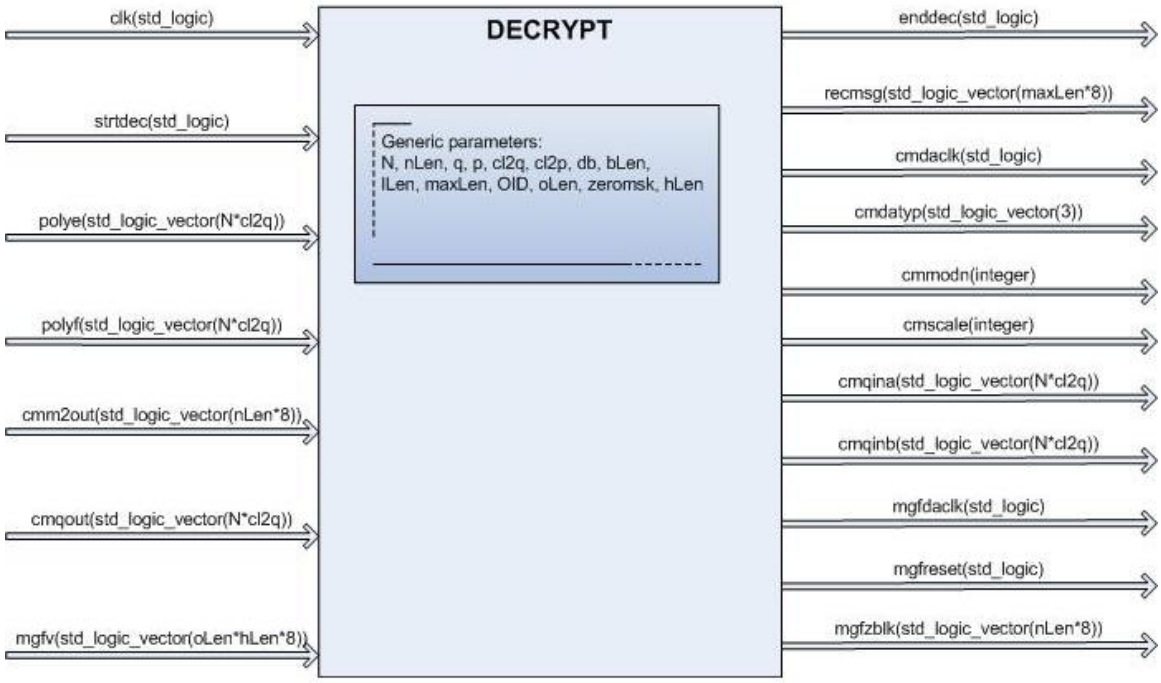

Figure 3.13: Block diagram for decryption operation

3.2.4 System model testing and results

The behavioral VHDL model described in Section 3.2.3 was mainly tested using data provided for the IEEE 1363.1 draft standard. The four parameter sets that data was available for were ees347ep2, ees397ep1, ees587ep1 and ees787ep1. Since testing data was not yet available for product forms, the focus of the testing was conducted using the less optimized full polynomial algorithms. The testbench used a series of constant assignments applied through the generics for each individual component model to control the parameters for testing.

```
--*******************************************************
-- EES587EP1
-------------------------------------------------------------
   constant N           : integer := 587;
   constant nLen        : integer := 74;     -- the number of bytes to hold N bits, nLen = ceil[N/8]
   constant IGFnLen     : integer := 2;      -- the number of bytes to hold the value N, IGFnLen = ceil[log2(N)/8]
   constant q           : integer := 439;
   constant p           : integer := 2;
   constant c12q        : integer := 14;
   constant c12p        : integer := 1;
   constant db          : integer := 192;
   constant bLen        : integer := 24;
   constant lLen        : integer := 1;
   constant maxLen      : integer := 48;
   constant dr          : integer := 108;
   constant dm0         : integer := 208;
   constant OID         : std_logic_vector(23 downto 0) := X"000501";
   constant zeromsk     : std_logic_vector(7 downto 0) := "00000111"; -- mask for the last (nLen*8)-N bits to set to zero
   constant sDataLen    : integer := 584;
   constant MGFnumblk   : integer := 2;      --size of input is nLen*8+32 bits - determine how many 512 blocks of input are needed including pa
                                             --use nzero := (448 - (nLen*8+32) - 1) mod 512; numblk = ceil[(nLen*8+32+1+nzero+64)/512];
   constant IGFnumblk   : integer := 2;      -- size of input is sDataLen+32 bits - determine how many 512 blocks of input are needed including
                                             --use nzero := (448 - (sDataLen+32) - 1) mod 512; numblk = (sDataLen+32+1+nzero+64)/512;
   constant hLen        : integer := 32;
   constant oLen        : integer := 3;
   constant oLenMin     : integer := 640;
   constant minCalls    : integer := 20;     -- minCalls is determined by (oLenMin*8)/hLen*8 where oLenMin is a provided system parameter
   constant IGFc        : integer := 11;

   constant cPeriod : TIME := 20 ns;

   -- Constants for input for ees587ep1
   constant ees587ep1l     : integer := 48;  --message length in bytes
   constant ees587ep1m     : std_logic_vector(ees587ep1l*8-1 downto 0) := X"6162636465616263646561626364656162636465616263646561626364656162636
   constant ees587ep1b     : std_logic_vector(db-1 downto 0) := X"07d4bd118ff1812df7e9d4ffaea177763335fc5e81ee2190";
   constant ees587ep1pF    : std_logic_vector(N*c12q-1 downto 0) := "00"&X"0000000000000000000000000000040000004000000040000000400100000000000000000
   constant ees587ep1g     : std_logic_vector(N*c12q-1 downto 0) := "00"&X"0000000000000004001000000100040010004000000000000040000004001000000100
   constant hmo4q          : integer := 8215;   -- highest multiple of 4 index for mod q polynomial (NOTE: subtract one for indexing)
--*******************************************************
```

Figure 3.14: Sample testbench setup for ees587ep1

Although the model was successfully verified for ess347ep2, ees397ep1 and ess587ep1, there were oddities found in the testing data which caused complications. The first unusual was found in the Index Generation Function (IGF) integer output values. Using the hex output values provided in the testing data, the hash algorithm output was verified and the difference was discovered. At the boundaries where one hash output ended and another began, an extra constant byte value was found to be inserted. The extra byte value required compensation in the indexing of the IGF function for both the SHA-1 and SHA-256 variants of the hash algorithm. Further complications developed when using a message that was less than the maximum message length as found in the ees787ep1 data set. Although the model created was able to handle messages of varying length, and was verified as being functional with messages less than the maximum message length, the testing for ess787ep1 did not match the expected results when generated indices using the IGF. Debugging using the hex value outputs of the hash algorithm was not possible in this case because none of the values matched. The message data being only one byte away from filling the maximum message length, every value for a byte being appended to the beginning or to the end of the message

was tried with no success. Although the testing data was very useful for verification of the IEEE 1363.1 draft standard model, it is clear that additional work is needed to clean up further test data sets.

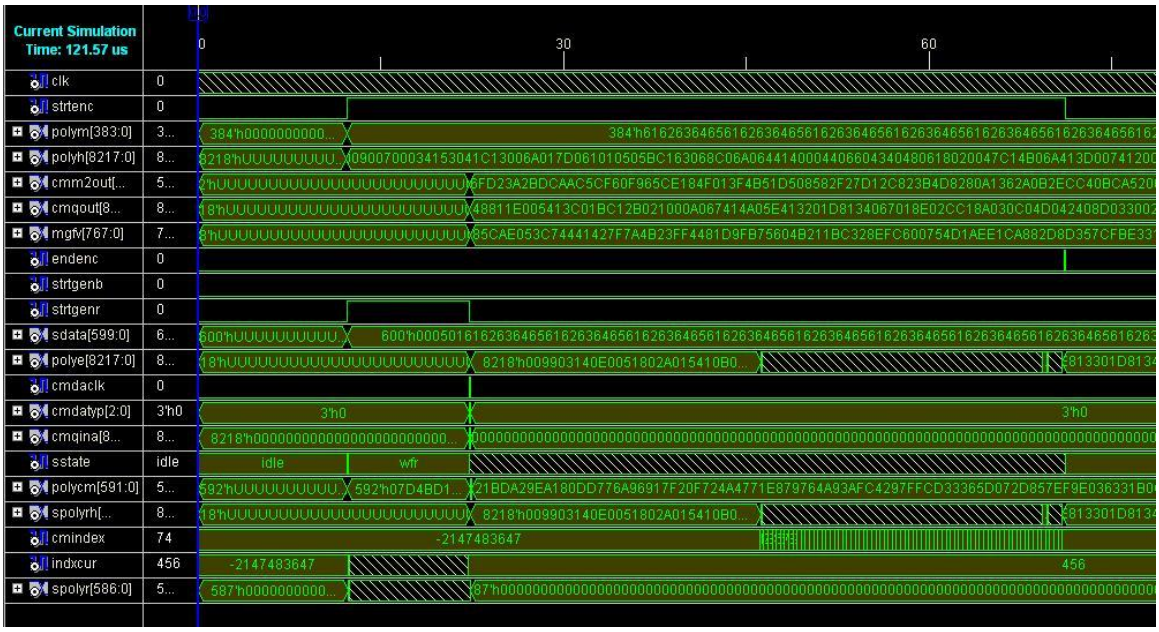

Figure 3.15: Sample waveform output for ees587ep1

Presented in Figure 3.15 are the results for the ees587ep1 test data set. In addition to the waveform data, text files were written for the private-key, public-key, ciphertext and recovered plaintext data for verification.

Chapter 4

Discussion and analysis

Throughout the investigation into the NTRU system and IEEE 1363.1 standard, a number of topics arose which required further analysis. Some of the topics related directly to the creation of the system model, while others were related to general issues in considering an NTRU system. The purpose of this chapter is to provide in depth discussion and analysis of select topics for which further consideration was needed.

4.1 System dependent design

Depending on the platform chosen to implement the NTRU system, a number of options are available on how to design different pieces of the system. If a combined hardware and software platform is available, there is a choice of what to implement in software and what is more efficient in hardware. Primarily, there exists a trade-off between the ease of implementation using software and potential higher performance using hardware in the convolution multiplication operation, polynomial inversion operation and the support algorithms of the IEEE 1363.1 draft standard. A discussion of a possible software method of performing the convolution multiplication operation is presented in section 4.4, but it is not clear that recent changes will not impact the performance of the algorithm or that hardware cannot achieve better results. Discussion of the inversion operation can be found in section 4.3. While inversion can be quickly and simplistically implemented in software, there is a performance advantage using a hardware implementation. The mask generation function

(MGF) and index generation function (IGF) support algorithms used in the IEEE 1363.1 draft standard are based on iterative calls to one of the SHA family of hash algorithms. The SHA hash algorithms, the MGF and the IGF are all easily developed using software, although better performance could be obtained through use of hardware implementations.

With respect to the total system, many of the choices for implementation of algorithms in software or hardware are dependent on the resources available. When resources are scarce, the entire system can be shifted towards a software implementation that will use minimal hardware to perform all operations while sacrificing performance. In resource abundant systems, a software version could obtain better performance through increased usage of resources but more likely it would be appealing to convert algorithms to dedicated hardware structures for increased performance. Since none of the pieces in the system are particularly challenging in either hardware or software, the design of each algorithm is flexible towards the requirements of the resources available. Sections 4.2 and 4.4 provide further discussion of design aspects that are related to the type and amount of resources available.

4.2 Storage methods and the relation to N and q

As mentioned in section 3.2.1, the method of storage and the amount of storage required to implement the NTRU system can be of particular concern for those using resource constrained hardware or for those seeking to optimize performance. The trivial method of storing polynomials is to store each coefficient in a linear array, taking $N * \lceil \log_2(q) \rceil$ or $N * \lceil \log_2(p) \rceil$ bits of storage for a polynomial modulo q or p respectively. The appeal of this method is the simplicity involved, however this is clearly not as appealing as the number of bits needed to store each coefficient or the degree of the polynomials increase. Using the idea that many of the coefficients involved in a polynomial will be zero, an alternative method is to store each non-zero coefficient and the degree that coefficient represents. Assuming there were num_{nz} percentage of non-zero coefficients, this method would require

$N * (\lceil \log_2(q) \rceil + \lceil \log_2(N) \rceil) * num_{nz}$ or $N * (\lceil \log_2(p) \rceil + \lceil \log_2(N) \rceil) * num_{nz}$ bits of storage per polynomial. To evaluate which method is better in a general manner is difficult to imagine because the equations are based on N, p or q and num_{nz}, which requires varying of three input variables and assessment of the output amount of storage. Instead, to avoid over complication during examination of the results, one of the three input variables can be constant, a second can be varied and a graph can be made for each of a series of values for the third value. The choice was made to hold a value for either p or q, vary N and graph the results for a series of different values for num_{nz}. To examine one of the extremes, the results for $p = 3$ are shown in Figure 4.1. The data points used to generate the storage graphs were taken from parameter sets found in [16].

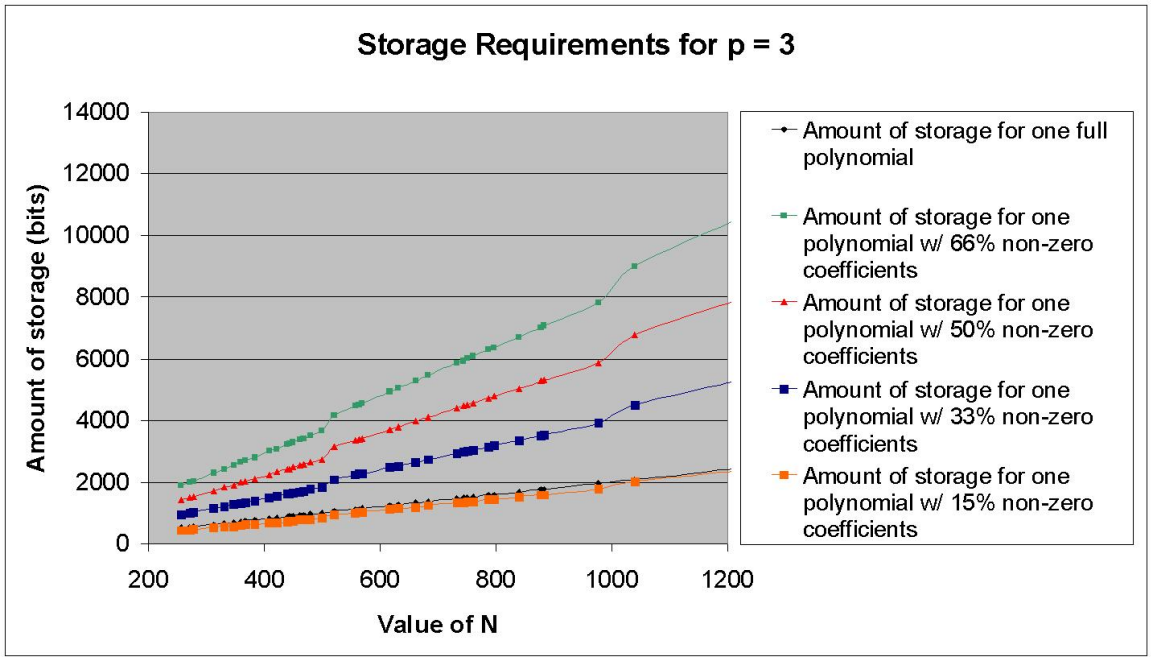

Figure 4.1: Polynomial storage for $p = 3$

In a case where the number of bits required to store the polynomial coefficients is much lower than the number of bits to store the index, the full storage method performs well enough that an alternative method is not necessary. A more reasonable comparison is found when comparing the two methods using polynomials modulo $q = 256$, as shown in

Figure 4.2.

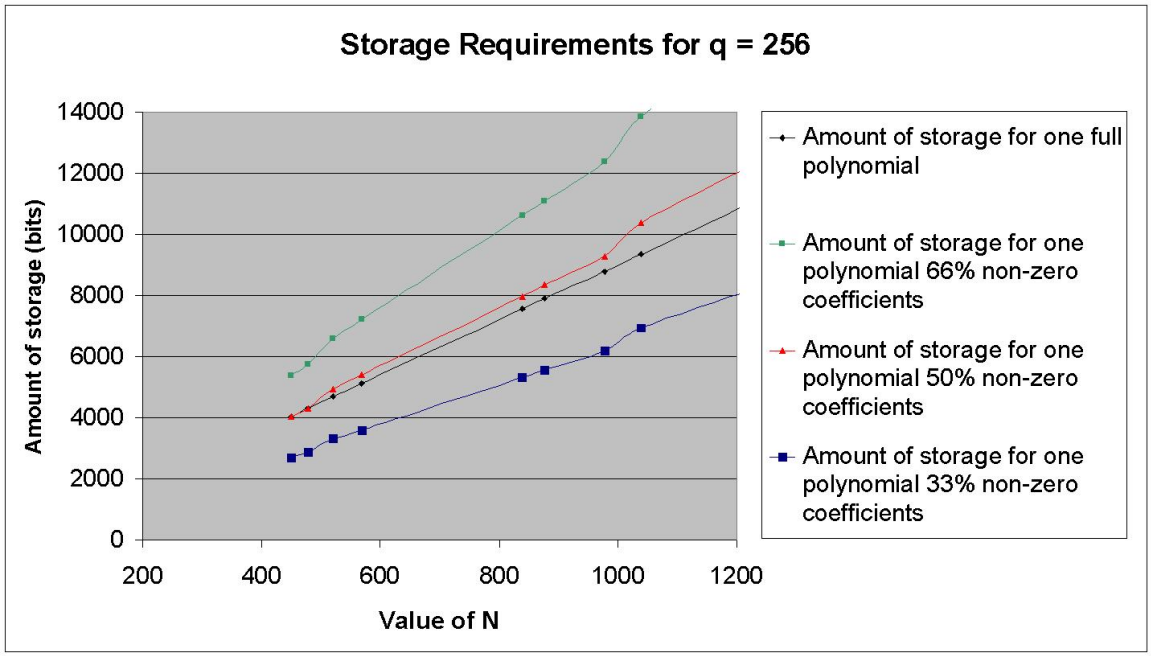

Figure 4.2: Polynomial storage for $q = 256$

The method of storing non-zero coefficients starts to become more appealing when polynomials are reduced using a larger modulus. To prevent imbalances in the polynomials which could lead to weaknesses to certain types of attacks, it is desirable that the polynomials have a somewhat balanced number of zero and non-zero coefficients. Checks implemented in the NTRU system usually use boundaries that a polynomial should contain between 33% and 66% non-zero coefficients. From Figure 4.2, it appears that the two methodologies are close to being equivalent, with the assumption that a general polynomial would have around 50% non-zero coefficients. However, an issue exists in that while the full storage methodology increases linearly with N, the method of storing only the non-zero coefficients increases approximately $O(N * \log_2(N))$. As a consequence, even if the methods are similar for lower degree polynomials, as N increases the methods will diverge and the full storage method will eventually becomes significantly more efficient. From a practical standpoint, it is not clear that polynomials of such a high degree would ever be

used, so the concern may be purely academic. Of clear significance, a comparison of the two methodologies for $q = 2048$ is shown in Figure 4.3.

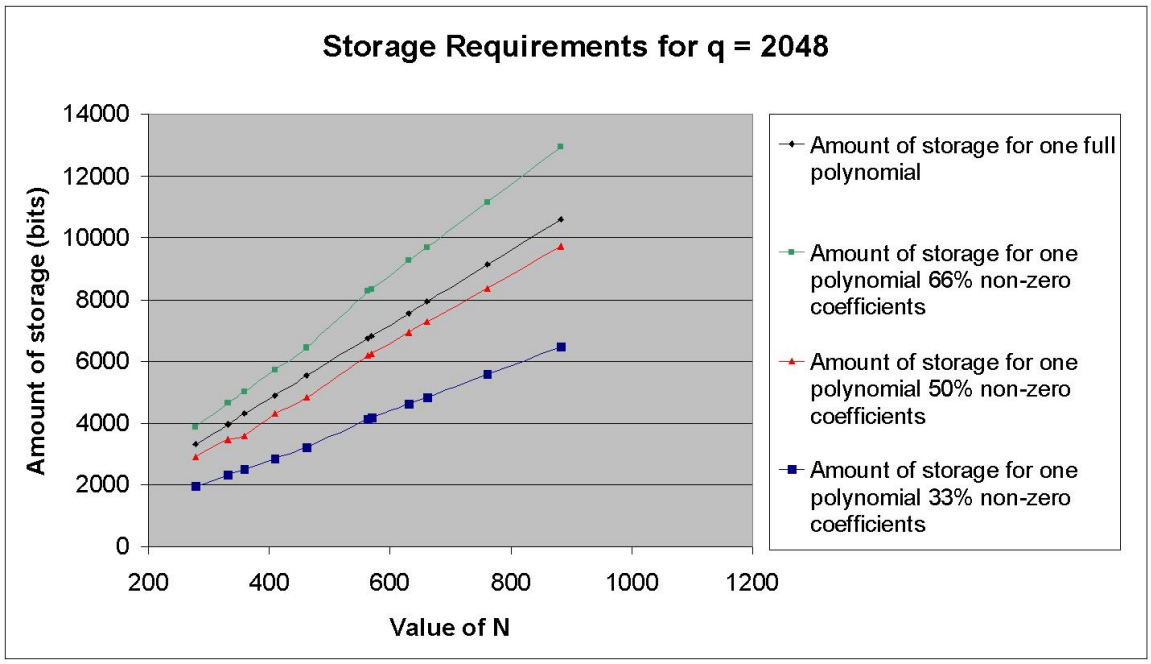

Figure 4.3: Polynomial storage for $q = 2048$

For this case, the method of storing the polynomials by their non-zero coefficients achieves slightly better results than storing the full polynomial. Indeed, the intersection of the two metholodogies can be found rather easily by setting the storage requirements equal to one another.

$$N * (\lceil \log_2(q) \rceil) = N * (\lceil \log_2(q) \rceil + \lceil \log_2(N) \rceil) * num_{nz}$$
$$\lceil \log_2(N) \rceil = \lceil \log_2(q) \rceil * \left(\frac{1}{num_{nz}} - 1 \right)$$

num$_{nz}$	33%	50%	66%
intersection point	$\approx 2 * (\lceil \log_2(q) \rceil)$	$= \lceil \log_2(q) \rceil$	$\approx \frac{1}{2} * (\lceil \log_2(q) \rceil)$
intersection for (N, q=2048)	$N \approx 4,194,304$	$N \approx 2048$	$N \approx 32$

To obtain a case where one representation performs dramatically better than the other, either $q << N$ or $N << q$. Rather than choose one method to represent all polynomials,

it would seem that it is better to dynamically choose the storage on whether the modulus that the polynomial is reduced by is much larger than or much smaller than the degree of the polynomial. If the reduction modulus and the degree are approximately the same, then either method could be used. In hardware, the method of storing non-zero coefficients and their degree may actually involve greater complexity and take longer to manage than storing the full polynomial. In such a situation, it would be appealing to have N and q be approximately the same or to have $N << q$. Having $N << q$ in hardware also has the appeal of reducing the clock cycles needed to iterate over a polynomial, which is much more costly than increasing the size of q. Overall, hardware would benefit by choosing the highest q value that fills the maximum bit operations supported and adjust the value of N to change the security level, optimally keeping the value of $N << q$. Software would have an easier time managing the polynomials stored by their non-zero coefficients and could choose to pick a value of q and drive the value of N higher to change the security level without concern for the polynomial storage method becoming less efficient.

The split between desiring $q << N$ or $N << q$ facilitates a discussion of the trade-off between increasing N as compared to increasing q. As a basic example, Figure 4.4 shows the trade-off of holding q constant while increasing N compared to holding N constant while increasing q from a starting point of $(N, q) = (251, 197)$.

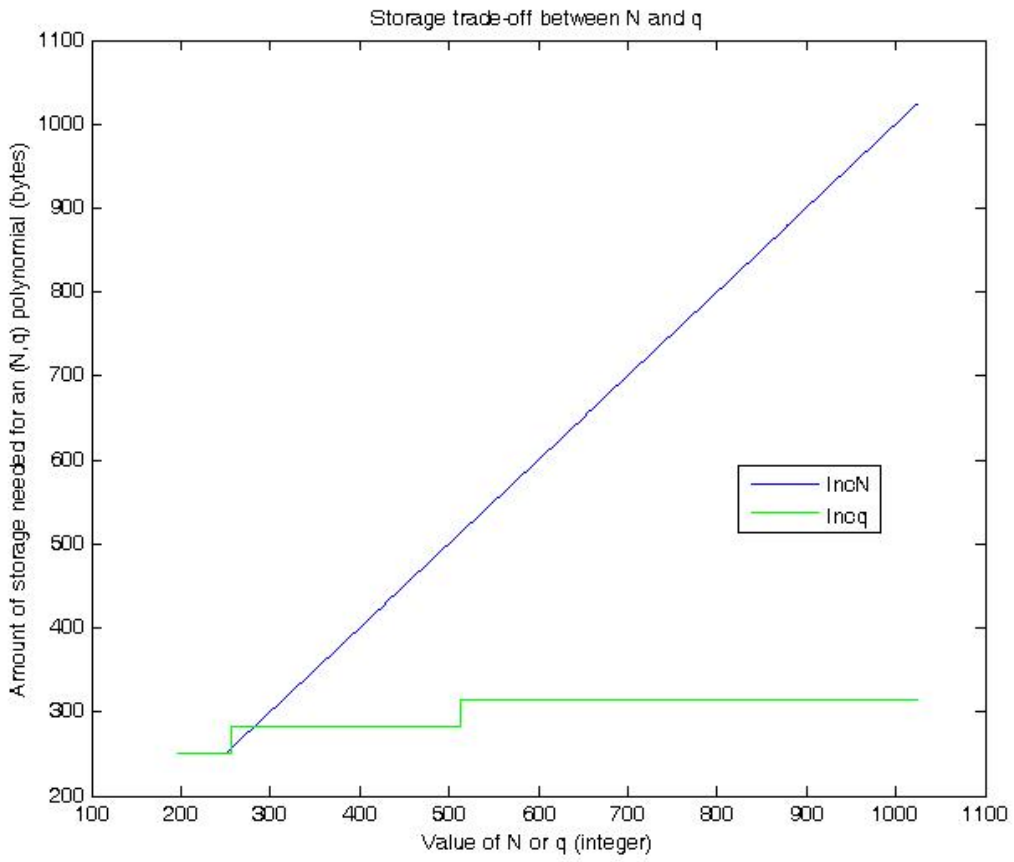

Figure 4.4: Trade-off in increasing value for $(N, q) = (251, 197)$

An initial look implies that the cost of increasing q is far less than the cost of increasing N, but the trade-off must also be considered with respect to the benefit in security. If the security is considered proportional to the search space required to brute force search for a matching polynomial, then an idea of the effect on security can be gained from estimating the search space of an (N, q) polynomial as the size of the coefficients multiplied by the number of coefficients, $N * q$. Now, either q or N can be incremented and the relative difference can be examined. While in practice this is completely invalid because N should always be a prime number and q and $q + 1$ most likely do not both fit the restraints on q, a change by one should be scalable to a general integer.

Polynomial space	**Estimate search space**	**Difference from baseline**
(N, q)	$N * q$	–
$(N + 1, q)$	$(N + 1) * q$	q
$(N, q + 1)$	$N * (q + 1)$	N

Table 4.1: Comparison of estimated search spaces

The effect on the search space when changing N is dependent on the value of q, while changes in q change the search space dependent on the value of N. If N and q are similar, then there should be little difference between increasing N or increasing q with respect to the search space. When $q << N$ or when $N << q$, then a situation of decaying returns exists until eventually increasing the higher valued variable any further will have comparitively small effect on the search space in relation to an increase of the smaller variable. A more specific relation between N, q and the lattice security of the system can be found in Annex A, section 2.2.7 of the IEEE 1363.1 draft standard [14]. There are two quantities introduced which are dependent on N and q and are used to experimentally determine the effect of changes on N and q with relation to the lattice security.

$$c = \rho * \sqrt{2N}$$
$$a = N/q$$

The specifics of these equations can be found in the standard, but the conclusion presented therein is that experimental results suggest that the lattice security is relative to the quantities c and a. In order to increase lattice breaking times, it is suggested that the ratio of N to q, a, should be held constant while increasing c. While this could be accomplished by increasing N, it requires increasing q as well to maintain the ratio a. An increase in both N and q causing an increase in security is consistent with the search space estimation used above, and confirms the idea that increasing only N or q will have diminishing returns. Additionally, it is stated that the experimental results found that hold c constant and increasing a causes a slight decrease in lattice breaking times. If c is held constant, N

must not be changing and therefore to increase a the value of q must be decreased, which again is consistent with the search space estimation. An assumption can be made that the opposite is true as well, holding c constant while decreasing a would slightly increase lattice breaking times. Such an assumption suggests that trying to achieve higher security levels through increases in q alone is not very efficient, but this is not entirely clear because the search space estimation method suggests that it would also depend on the value of q relative to N. Tracing the discussion back to [22], then it is further suggested that holding a and c constant while increasing N yields increases in lattice breaking times. The results shown illustrate the effect of increasing N, but not the case which is appealing to hardware of holding N constant while increasing q. For further study and experimentation, the parameter generation algorithm shown in [16] could be used.

4.3 Polynomial inversion

Inversion is an expensive operation in many cryptosystems, and such is the case for the NTRU system. For key generation in NTRU, the inverse of the private key polynomial must be taken modulo the large modulus, q. For security purposes, it is also checked that the inverse of g exists modulo q. For now, two suggested algorithms exist for calculation of the inverse, the Extended Euclidean Algorithm (EEA) and the Almost Inverse Algorithm (AIA), of which only the EEA is recommended for use in the IEEE 1363.1 draft standard. Although an implementation of each algorithm was tested during the course of this research, they were not developed to the point where a reasonable comparison of hardware results could be made. Instead, a comparison can be made by examining one iteration of each algorithm in terms of the operations required. To start with, the general EEA presented in Algorithm 3.3 can be compared with the general AIA presented in Algorithm 3.4.

Operation	# in EEA	# in AIA	Steps found in
Integer inversion	1	1	*EEA: Step 8* *AIA: Step 10*
Polynomial rotation	0	2	*AIA: Step 4*
Polynomial degree	1	2	*EEA: Step 9* *AIA: Steps 5 and 8*
Polynomial addition/subtraction	1/2	0/2	*EEA: Step 13 / Steps 12 and 14* *AIA: Steps 11 and 12*
Polynomial convolution multiplication	2	2	*EEA: Steps 12 and 14* *AIA: Steps 11 and 12*

Table 4.2: Comparison of operations per iteration between inversion methods

From the comparison in Table 4.2, it might be thought that the Almost Inverse Algorithm contains more work per iteration than the Extended Euclidean Algorithm. The AIA has two polynomial rotations and one polynomial degree calculation more than the EEA, but one fewer polynomial addition. Note that the caclulation in *Step 11* of the EEA appears to have a polynomial rotation, but this is actually just an assignment of one coefficient of the polynomial v. In addition, there is final work done in *Step 20* of the EEA and in *Steps 6* and *7* of the AIA, of which both have an integer inversion and polynomial scale operation but the AIA has an additional polynomial rotation operation. While this could be considered a fair comparison of the main loops of each algorithm, what it ignores is the inner loop inside the EEA. The inner loop of the EEA contains a polynomial degree calculation, a polynomial addition, a polynomial subtraction and a polynomial convolution multiplication. Assuming the inner loop executes at least twice on a given iteration of the EEA, the AIA achieves inversion through fewer computations.

When working with the inverse, considerations should be made for raising a polynomial inverse result modulo a prime to a power of a prime, per the recommendations of [16]. Using Algorithm 3.2, a result can be raised to a power of a prime in approximately $\log_2(r)$

steps for a polynomial modulo p^r. Per iteration, there are two polyomial convolution multiplication operations, one polynomial scaling and one polynomial subtraction. The polynomial scaling can be achieved at almost no cost in hardware because it is a multiplication by two. Further savings can be achieved by observing that a polynomial convolution squaring may be less computationally expensive than a polynomial convolution multiplication. If N is an odd prime, then the coefficients of the square of a polynomial $a(X)$ can be computed as follows, where the subscripts shown should be taken modulo N.

$$a(X)_k^2 = a_{N-2k}^2 + 2(a_{N-2k-1} \cdot a_{N-2k+1}) + 2(a_{N-2k-2} \cdot a_{N-2k+2}) + ...$$
$$+2(a_{N-2k-\frac{N-2}{2}} \cdot a_{N-2k+\frac{N-2}{2}}) + 2(a_{N-2k-\frac{N-1}{2}} \cdot a_{N-2k+\frac{N-1}{2}})$$

Using the basic convolution multiplication operation, a squaring would take approximately N^2 operations. With the squaring equation shown above, the inner loop of the equation can be reduced to approximately $N/2$ operations, leading to an overall number of approximately $\frac{N^2}{2}$ operations.

4.4 Considerations for parallelism

The NTRU system has many pieces which can be implemented in either a serial style or a parallel style. In addition to adjusting system parameters to provide flexibility in security and performance, this also means that the calculations themselves can be adjusted to provide flexibility in performance. Essentially all of the polynomial operations require independent iterative calculations on the polynomial operands, lending themselves to parallel implementation. When designing a system around the IEEE 1363.1 draft standard, additional parallelism can be achieved in the support algorithms.

Polynomial addition, subtraction and modular reduction all require an iteration through all of the coefficients of one or more polynomials, calculating results for each degree which are independent of each other. Any or all of these can be executed simultaneously through the instantiation of multiple pieces of hardware to calculate the result. An additional technique that could be used with the addition or subtraction of polynomials is the computation

of multiple output coefficients using the same addition or subtraction unit. For instance, if the unsigned coefficients of a polynomial are represented using 8 bits, then two coefficients could be concatenated together and used as inputs to a 16 bit addition unit. A similar method could be used for subtraction, however the carry would not be propagated between coefficient boundaries.

Parallelism for polynomial convolution multiplication can easily be achieved as well. In [11], a design for a convolution multiplier with flexibility in the number of coefficients that can be computed simultaneously is introduced.

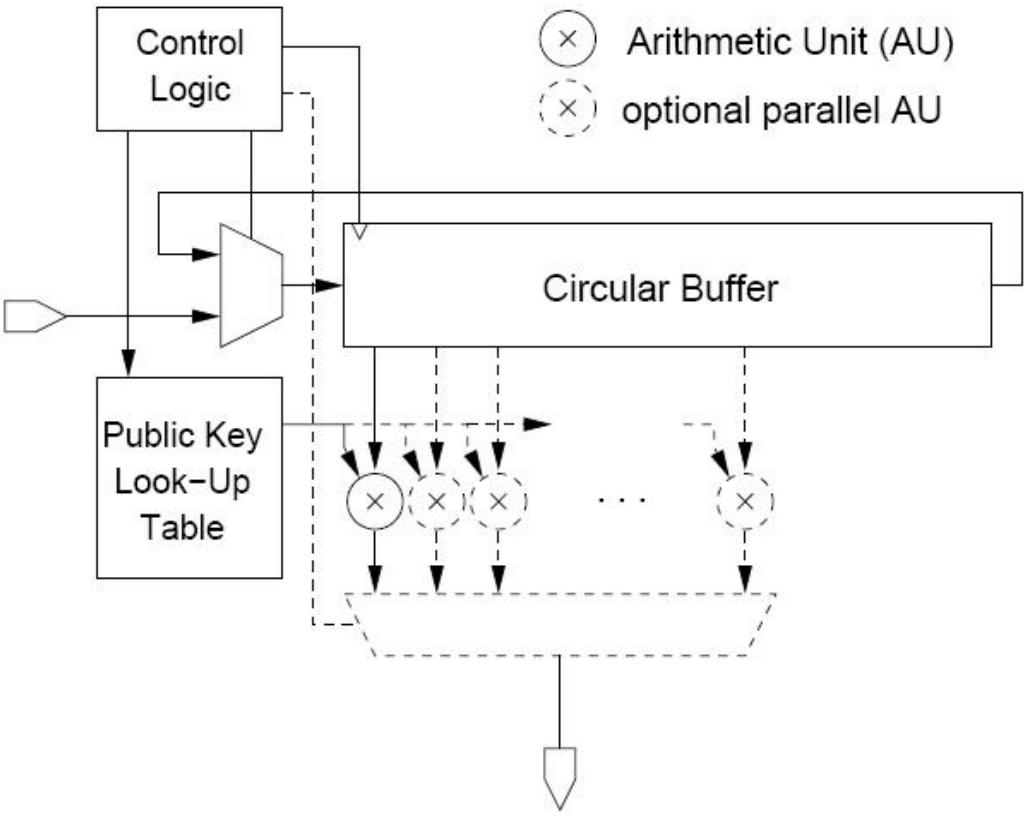

Figure 4.5: Scalable convolution multiplier [11]

The design allows loading of a modulo p operand through a multiplexer into a circular shift register, while a modulo q operand is stored in a look-up table. The control logic is designed to control each arithmetic unit to calculate one coefficient of the result in $N + 8$

clock cycles, for a total of $N * (N + 8)$ clock cylcles for a complete convolution multiplication with one arithmetic unit. The total number of clock cycles taken for multiple arithmetic units is $(N + 8)(\lceil N/k \rceil) + k - 1$ for k arithmetic units. The design is appealing because of the flexibility in calculating a single or multiple output coefficients at once, but there are some easily changed drawbacks that are caused by the restrictive nature of the system the multiplier was designed for. To save storage room, the multiplier uses a look-up table for one of the operands, which could easily be modified to use a memory block. The restriction on multiplying only a modulo p polynomial by a modulo q polynomial can be modified by extending the circular shift register to fit a modulo q polynomial. This would affect the arithmetic units as well but should not change the control logic other than the number of bits that the circular shift register is moved when changing coefficients. The last modification that would be appealing is to add in the ability to input a scalar quantity that can be multiplied with each output coefficient for computation of results similar to the creation of the public key, $h = pf_q * g \pmod{q}$. A final comment on the design is that it does not efficiently handle the number of zero coefficients that are found in the operands. In IEEE 1363.1, it is almost gauranteed that the input operands will have between 33% and 66% non-zero coefficients, which means that significantly better speed might be achieved by ignoring calculations in which one of the input coefficients is zero.

An alternative approach is taken in [26], which also can be used to exploit parallelism. Figure 4.6 shows the general concept of computing a convolution multiplication through the use of only additions and rotations of the modulo q input polynomial.

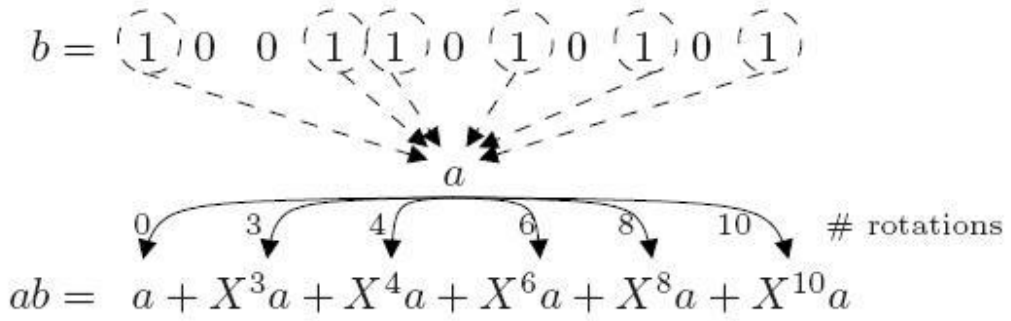

Figure 4.6: a*b using polynomial additions and rotations [26]

By examining the patterns that are found in the coefficients of the modulo p operand, partial results can be computed and reused if the same bit pattern is found in the modulo p operand again. Figure 4.7 shows an example wherein the partial result $(a + X^2a)$ is used twice to compute the output $a * b$. In comparison to the method which does not use bit patterns, this saves one addition and one rotation. If the patterns are found and the partial results are calculated simultaneously, this method achieves significant speed advantage over the naive nested loop method of computing the product.

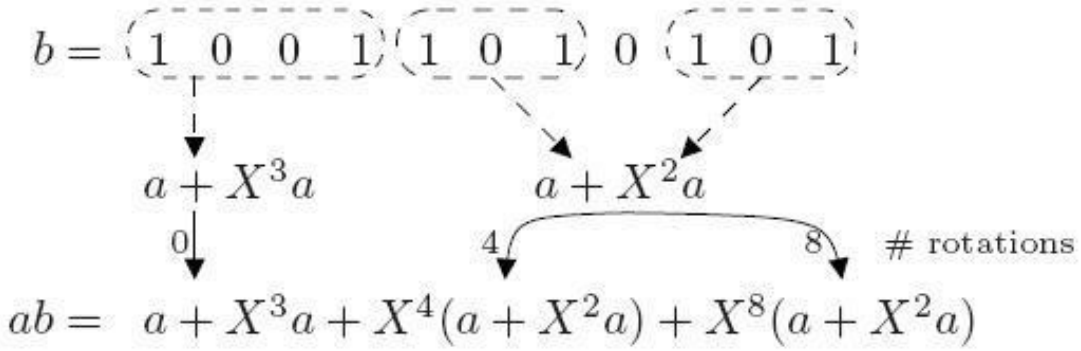

Figure 4.7: a*b using bit patterns [26]

While this is effective using binary polynomials, the chances of finding the same pattern multiple times in a ternary polynomial are not as good. Although this would reduce the efficiency of the algorithm, testing needs to be done to determine the impact.

The mask generation function (MGF) and index generation function (IGF) of the IEEE 1363.1 draft standard call a hash function a number of calculated times. The input to the hash algorithm is fed the same input data each time except that a counter is appended and incremented during each iteration. Since the output of each successive call is not dependent on the cycle before, the hash outputs could be calculated in parallel. For lower security parameter sets, the MGF is often only called once and each hash output of the IGF is used to calculate multiple indices, meaning that parallelism would not provide much, if any, performance benefit. Higher security sets, which would use larger values for N and q, might require multiple hash outputs from the MGF and each hash output of the IGF may provide few indices, perhaps allowing parallelism to provide significant benefits. Since higher security parameter sets would most likely be used on high performance systems, the cost of implementing simultaneous hash algorithms would most likely be minimal compared to the available hardware resources.

A final consideration for parallelism can be found with respect to key generation in the IEEE 1363.1 draft standard. During key generation, calculation and inversion of the private key, f, can be done while the random polynomial, g, is generated, inverted and checked for suitability. Two sets of inversion hardware, or pipelining of one set of inversion hardware, would allow the calculations to be more efficiently completed.

Chapter 5

Conclusion

Presented in this thesis is a parameter and component flexible testing model for the NTRU Public-Key Cryptosystem in conformance to the IEEE 1363.1 draft standard. The model was successfully tested using provided and generated test data sets and is now adaptable for further software and hardware research. Research conducted during creation and testing of the model was used to analyze the NTRU system with respect to both general underlying mathematical operations and specific qualities relating to the IEEE 1363.1 draft standard. The results of the research suggest that the system is highly adaptable to many conditions based on choices in the system parameters. Representation of the polynomial operands can be chosen to maximize storage efficiency. Hardware implementations benefit by using the maximum value of q which fits the bit width allowable in the hardware, followed by adjustment in N to achieve the desired security level. Parallelism in the operations of the system can be exploited to achieve better efficiency but must be carefully considered to avoid complications when adapting to new parameter sets. Overall, the work presented here can be used as underlying research for further investigation of the NTRU system and IEEE 1363.1 draft standard.

5.1 Future of NTRU

Although the NTRU Public-key Cryptosystem has been around for a short period of time compared to other classical cryptosystems, it seems to have enough support to be a lasting entity. Due to the system being in the early stages, the recommended implementation characteristics and suggested practices are still changing rapidly but the underlying mathematical concepts do not appear to be in question. Implementation of NTRU in resource constrained hardware is an active area of development and may lead to usage in a number of wired and wireless applications. Discussion has also begun about the advantages of the NTRU system compared to classical systems in a quantum computing environment. Although the integer factorization and discrete logarithm problems that many classical cryptosystems are based on have known efficient algorithms for solving in a quantum computing environment, it is unclear whether such an algorithm will be found to solve the closest vector and shortest vector problems that are used in the NTRU cryptosystem. Certainly, additional in depth research will need to be conducted and circulated in order to further evaluate the NTRU system.

5.2 Future work

Although previous research efforts have been tightly focused, the opportunities for further research are quite extensive. Extensions to the work presented in this thesis can be done in either software, hardware or a mixture of both. Software work might continue with further investigation of product form polynomials. Additional component focused work could be done to explore new methods and optimize performance of existing components, of which the polynomial convolution multiplication and inversion components are most likely of highest interest. An exploration of optimizations over the software/hardware boundary could be conducted to seek optimal performance or a purely hardware solution might be sought through conversion of the current model to purely structural code.

More general topics of research may include additional research into the possibility of

modified or new parameter sets and integration of NTRU with a private-key scheme. Parameters sets which use higher values for p which could fit a nibble or byte worth of data may be appealing instead of adapting binary information to ternary representation. An efficient implementation of NTRU alongside a private-key scheme could provide a maximum efficiency complete cryptographic solution for communication establishment and continuous data transfer.

Bibliography

[1] D. R. Stinson, *Cryptography: Theory and Practice*, 3rd ed. Chapman & Hall/CRC, 2006.

[2] A. Kerckhoffs, "La cryptographie militaire," in *Journal des sciencas militaires* vol. 9, 1883, pp. 5–83.

[3] C. E. Shannon and W. Weaver, *The Mathematical Theory of Communication*. University of Illinois Press, 1963.

[4] W. Diffie and M. E. Hellman, "New directions in cryptography," in *IEEE Transactions on Information Theory*, 1976, pp. 644–654.

[5] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," in *Communications of the ACM,* vol. 21, 1978, pp. 120–126.

[6] T. ElGamal, "A public-key cryptosystem and a signature scheme based on discrete logarithms," in *IEEE Transactions on Information Theory* vol. 31, 1985, pp. 469–472.

[7] N. Koblitz, "Elliptic curve cryptosystems," in *Mathematics of Computation* vol. 48, 1987, pp. 203–209.

[8] V. S. Miller, "Use of elliptic curves in cryptography," in *CRYPTO 85*, 1985.

[9] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler, "Spins: security protocols for sensor networks," in *Wireless Networks*, 2002, pp. 521–534.

[10] K. Ren, K. Zeng, and W. Lou, "On broadcast authentication in wireless networks," in *International Conference on Wireless Algorithms, Systems, Applications (WASA 2006)*, 2006.

[11] J.-P. Kaps, "Cryptography for ultra-low power devices," Ph.D. dissertation, Worcester Polytechnic Institute, 2006.

[12] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A ring-based public key cryptosystem," in J. P. Buhler, editor, *Algorithmic Number Theory (ANTS III),* volume 1423 of *Lecture Notes in Computer Science (LNCS)*, 1998, p. 267288.

[13] Consortium for Efficient Embedded Security, "Efficient embedded security standard (EESS) #1," 2003. [Online]. Available: `http://www.ceesstandards.org`

[14] IEEE P1363 Working Group for Standards In Public Key Cryptography, "IEEE P1363.1/D9 Draft Standard for Public-Key Cryptographic Techniques Based on Hard Problems over Lattices." Institute of Electrical and Electronics Engineers, Inc., 2007. [Online]. Available: `http://grouper.ieee.org/groups/1363/lattPK/`

[15] N. Howgrave-Graham, "A hybrid lattice-reduction and meet-in-the-middle attack against NTRU." NTRU Cryptosystems, Inc., 2007.

[16] J. Hoffstein, N. Howgrave-Graham, J. Pipher, J. H. Silverman, and W. Whyte, "Hybrid lattice reduction and meet in the middle resistant parameter selection for NTRUEncrypt." NTRU Cryptosystems, Inc., 2007.

[17] IEEE P1363 Working Group for Standards In Public Key Cryptography, "IEEE 1363-2000 Standard Specifications for Public-Key Cryptography." Institute of Electrical and Electronics Engineers, Inc., 2000. [Online]. Available: `http://grouper.ieee.org/groups/1363/P1363/`

[18] IEEE P1363 Working Group for Standards In Public Key Cryptography, "IEEE 1363a-2004 Standard Specifications for Public-Key Cryptography - Amendment 1: Additional Techniques." Institute of Electrical and Electronics Engineers, Inc., 2004. [Online]. Available: `http://grouper.ieee.org/groups/1363/P1363a/`

[19] IEEE P1363 Working Group for Standards In Public Key Cryptography, "IEEE P1363.2/D26 Draft Standard for Standard Specifications for Password-Based Public-Key Cryptographic Techniques." Institute of Electrical and Electronics Engineers, Inc., 2006. [Online]. Available: `http://grouper.ieee.org/groups/1363/passwdPK/`

[20] IEEE P1363 Working Group for Standards In Public Key Cryptography, "IEEE P1363.3/D0.7.1 Draft Standard for Identity-Based Public-Key Cryptography."

Institute of Electrical and Electronics Engineers, Inc., 2006. [Online]. Available: http://grouper.ieee.org/groups/1363/IBC/

[21] "Federal information processing standards publication 180-2: Secure hash standard." National Institute of Science and Technology (NIST), 2002. [Online]. Available: http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

[22] J. Hoffstein, J. H. Silverman, and W. Whyte, "NTRU cryptosystems technical report #12, version 2: Estimated breaking times for NTRU lattices." NTRU Cryptosystems, Inc., 2003. [Online]. Available: http://ntru.com/

[23] N. Howgrave-Graham, J. H. Silverman, and W. Whyte, "Choosing parameter sets for NTRUEncrypt with NAEP and SVES-3." NTRU Cryptosystems, Inc., 2005. [Online]. Available: http://ntru.com/

[24] D. V. Bailey, D. Coffin, A. Elbirt, J. H. Silverman, and A. D. Woodbury, "NTRU in constrained devices," in *Workshop on Cryptographic Hardware and Embedded Systems - CHES 2001*, 2001, pp. 266–277.

[25] C. M. O'Rourke, "Efficient NTRU implementations," Master's thesis, Worcester Polytechnic Institute, 2002.

[26] J. Buchmann, M. Döring, and R. Lindner, "Efficiency improvement for NTRU." Technische Universität Darmstadt, 2007.

[27] P. L. Montgomery, "Modular multiplication without trial division," in *Mathematics of Computation* vol. 44, 1985, pp. 519–521.

[28] J. H. Silverman, "NTRU cryptosystems technical report #14, version 1: Almost inverses and fast NTRU key creation." NTRU Cryptosystems, Inc., 1999. [Online]. Available: http://ntru.com/

[29] J. H. Silverman, "NTRU cryptosystems technical report #10, version 1: High-speed multiplication of (truncated) polynomials." NTRU Cryptosystems, Inc., 1999. [Online]. Available: http://ntru.com/

[30] J. Hoffstein and J. H. Silverman, "Random small hamming weight products with applications to cryptography." NTRU Cryptosystems, Inc., 2000. [Online]. Available: http://ntru.com/

[31] J. H. Silverman and W. Whyte, "NTRU cryptosystems technical report #21, version 1: Timing attacks on NTRUEncrypt via variation in the number of hash calls." NTRU Cryptosystems, Inc., 2006. [Online]. Available: http://ntru.com/

[32] "Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family." National Institute of Standards and Technology (NIST), 2007. [Online]. Available: http://csrc.nist.gov/groups/ST/hash/sha-3/index.html